FreeBSD™ JOURNAL

May/June 2014 Vol. 1 • Issue No. 3

Networking

DTrace NEW PROBES

KQUEUE MADNESS

TCP SCALING

IPFW OVERVIEW

DNSSEC DATA DISTRIBUTION

iXsystems®

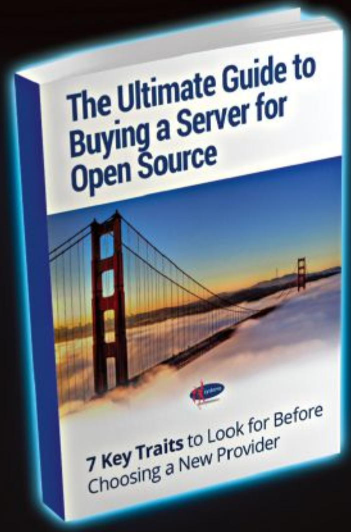
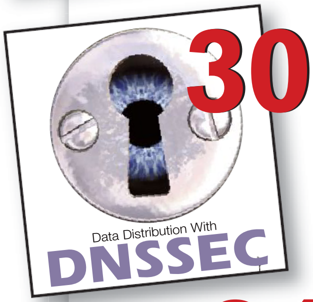
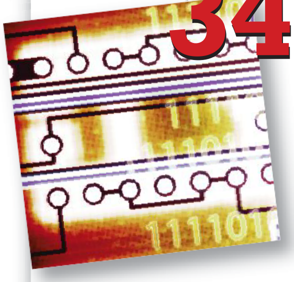

The Ultimate Guide to Buying a Server for Open Source
7 Key Traits to Look for Before Choosing a New Provider

Table of Contents

BY MARK JOHNSTON
DTrace
FreeBSD offers a plethora of tools and tricks for answering questions about activity in the network stack.





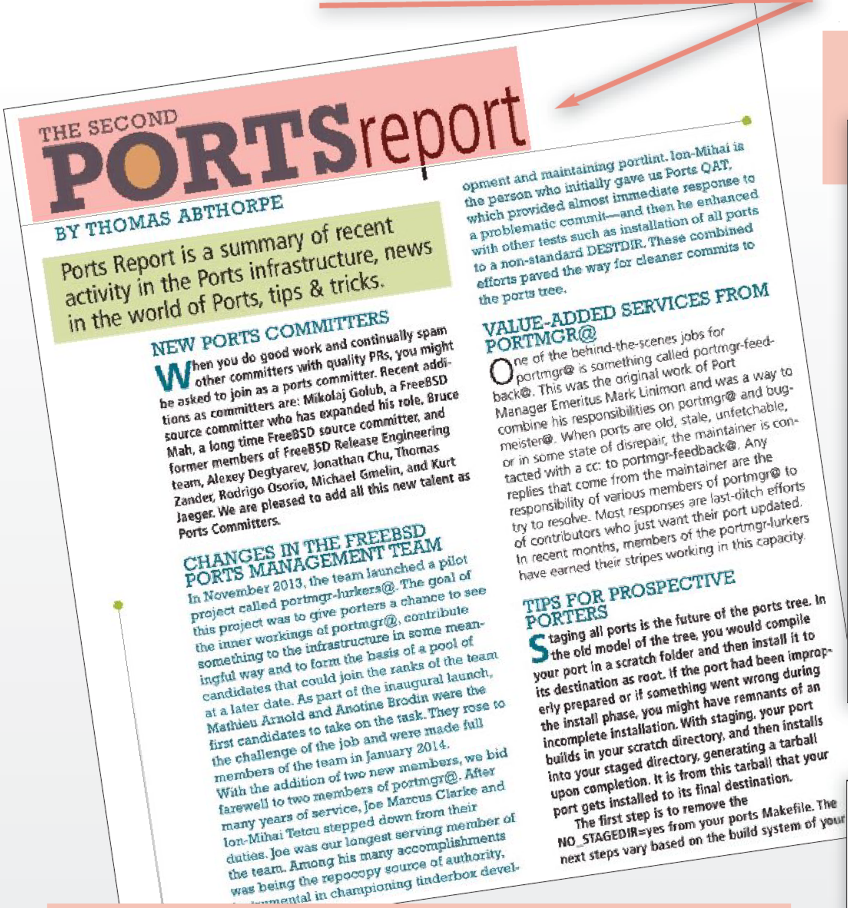
COLUMNS & DEPARTMENTS

APP FEATURES™

YOUR QUESTIONS ANSWERED

Q: Can I view the issue content as "text only"?

A: Yes. **Tap on highlighted title to see the "text only" screen.**



THE SECOND **PORTS**report

BY THOMAS ABTHORPE

Ports Report is a summary of recent activity in the Ports infrastructure, news in the world of Ports, tips & tricks.

NEW PORTS COMMITTERS

When you do good work and continually spam other committers with quality PRs, you might be asked to join as a ports committer. Recent additions as committers are: Mikolaj Golub, a FreeBSD source committer who has expanded his role, Bruce Mah, a long time FreeBSD source committer, and former members of FreeBSD Release Engineering team, Alexey Degtyarev, Jonathan Chu, Thomas Zander, Rodrigo Osorio, Michael Gmelin, and Kurt Jaeger. We are pleased to add all this new talent as Ports Committers.

CHANGES IN THE FREEBSD PORTS MANAGEMENT TEAM

In November 2013, the team launched a pilot project called portmgr-lurkers@. The goal of this project was to give porters a chance to see the inner workings of portmgr@, contribute something to the infrastructure in some meaningful way and to form the basis of a pool of candidates that could join the ranks of the team at a later date. As part of the inaugural launch, Mathieu Arnold and Anotine Brodin were the first candidates to take on the task. They rose to the challenge of the job and were made full members of the team in January 2014. With the addition of two new members, we bid farewell to two members of portmgr@. After many years of service, Joe Marcus Clarke and Ion-Mihai Tetcu stepped down from their duties. Joe was our longest serving member of the team. Among his many accomplishments was being the repocopy source of authority, instrumental in championing tinderbox development and maintaining portlint. Ion-Mihai is the person who initially gave us Ports QAT, which provided almost immediate response to a problematic commit—and then he enhanced with other tests such as installation of all ports to a non-standard DESTDIR. These combined efforts paved the way for cleaner commits to the ports tree.

VALUE-ADDED SERVICES FROM PORTMGR@

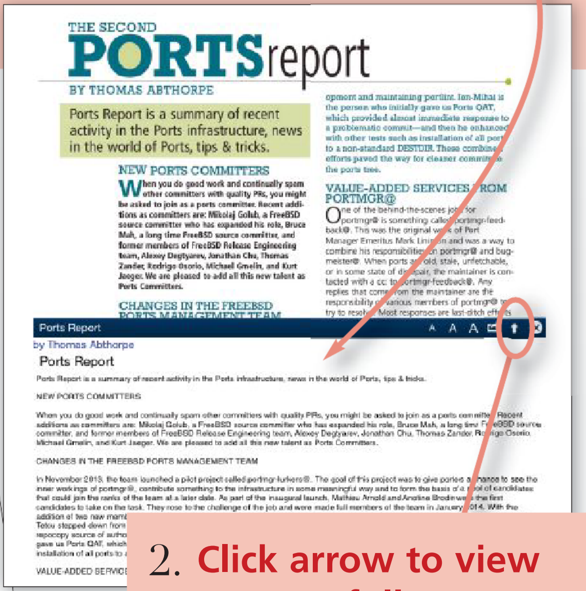
One of the behind-the-scenes jobs for portmgr@ is something called portmgr-feedback@. This was the original work of Port Manager Emeritus Mark Linimon and was a way to combine his responsibilities on portmgr@ and bugmeister@. When ports are old, stale, unfetchable, or in some state of disrepair, the maintainer is contacted with a cc: to portmgr-feedback@. Any replies that come from the maintainer are the responsibility of various members of portmgr@ to try to resolve. Most responses are last-ditch efforts of contributors who just want their port updated. In recent months, members of the portmgr-lurkers have earned their stripes working in this capacity.

TIPS FOR PROSPECTIVE PORTERS

Staging all ports is the future of the ports tree. In the old model of the tree, you would compile your port in a scratch folder and then install it to its destination as root. If the port had been improperly prepared or if something went wrong during the install phase, you might have remnants of an incomplete installation. With staging, your port builds in your scratch directory, and then installs into your staged directory, generating a tarball upon completion. It is from this tarball that your port gets installed to its final destination.

The first step is to remove the NO_STAGEDIR=yes from your ports Makefile. The next steps vary based on the build system of your

1. **The "text only" screen.**



THE SECOND **PORTS**report

BY THOMAS ABTHORPE

Ports Report is a summary of recent activity in the Ports infrastructure, news in the world of Ports, tips & tricks.

NEW PORTS COMMITTERS

When you do good work and continually spam other committers with quality PRs, you might be asked to join as a ports committer. Recent additions as committers are: Mikolaj Golub, a FreeBSD source committer who has expanded his role, Bruce Mah, a long time FreeBSD source committer, and former members of FreeBSD Release Engineering team, Alexey Degtyarev, Jonathan Chu, Thomas Zander, Rodrigo Osorio, Michael Gmelin, and Kurt Jaeger. We are pleased to add all this new talent as Ports Committers.

CHANGES IN THE FREEBSD PORTS MANAGEMENT TEAM

opment and maintaining portlint. Ion-Mihai is the person who initially gave us Ports QAT, which provided almost immediate response to a problematic commit—and then he enhanced with other tests such as installation of all ports to a non-standard DESTDIR. These combined efforts paved the way for cleaner commits to the ports tree.

VALUE-ADDED SERVICES FROM PORTMGR@

One of the behind-the-scenes jobs for portmgr@ is something called portmgr-feedback@. This was the original work of Port Manager Emeritus Mark Linimon and was a way to combine his responsibilities on portmgr@ and bugmeister@. When ports are old, stale, unfetchable, or in some state of disrepair, the maintainer is contacted with a cc: to portmgr-feedback@. Any replies that come from the maintainer are the responsibility of various members of portmgr@ to try to resolve. Most responses are last-ditch efforts

by Thomas Abthorpe

Ports Report

Ports Report is a summary of recent activity in the Ports infrastructure, news in the world of Ports, tips & tricks.

NEW PORTS COMMITTERS

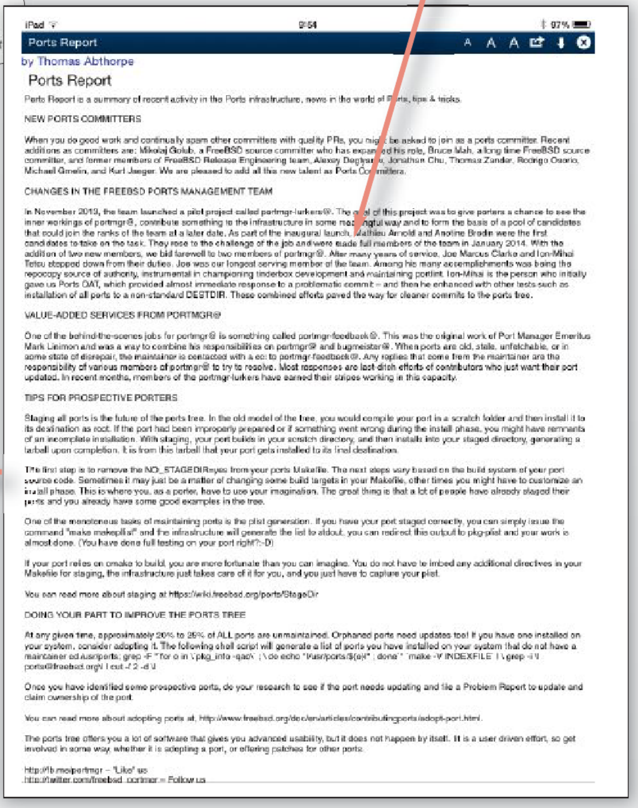
When you do good work and continually spam other committers with quality PRs, you might be asked to join as a ports committer. Recent additions as committers are: Mikolaj Golub, a FreeBSD source committer who has expanded his role, Bruce Mah, a long time FreeBSD source committer, and former members of FreeBSD Release Engineering team, Alexey Degtyarev, Jonathan Chu, Thomas Zander, Rodrigo Osorio, Michael Gmelin, and Kurt Jaeger. We are pleased to add all this new talent as Ports Committers.

CHANGES IN THE FREEBSD PORTS MANAGEMENT TEAM

In November 2013, the team launched a pilot project called portmgr-lurkers@. The goal of this project was to give porters a chance to see the inner workings of portmgr@, contribute something to the infrastructure in some meaningful way and to form the basis of a pool of candidates that could join the ranks of the team at a later date. As part of the inaugural launch, Mathieu Arnold and Anotine Brodin were the first candidates to take on the task. They rose to the challenge of the job and were made full members of the team in January 2014. With the addition of two new mem

VALUE-ADDED SERVIC

2. **Click arrow to view text on full screen.**



by Thomas Abthorpe

Ports Report

Ports Report is a summary of recent activity in the Ports infrastructure, news in the world of Ports, tips & tricks.

NEW PORTS COMMITTERS

When you do good work and continually spam other committers with quality PRs, you might be asked to join as a ports committer. Recent additions as committers are: Mikolaj Golub, a FreeBSD source committer who has expanded his role, Bruce Mah, a long time FreeBSD source committer, and former members of FreeBSD Release Engineering team, Alexey Degtyarev, Jonathan Chu, Thomas Zander, Rodrigo Osorio, Michael Gmelin, and Kurt Jaeger. We are pleased to add all this new talent as Ports Committers.

CHANGES IN THE FREEBSD PORTS MANAGEMENT TEAM

In November 2013, the team launched a pilot project called portmgr-lurkers@. The goal of this project was to give porters a chance to see the inner workings of portmgr@, contribute something to the infrastructure in some meaningful way and to form the basis of a pool of candidates that could join the ranks of the team at a later date. As part of the inaugural launch, Mathieu Arnold and Anotine Brodin were the first candidates to take on the task. They rose to the challenge of the job and were made full members of the team in January 2014. With the addition of two new members, we bid farewell to two members of portmgr@. After many years of service, Joe Marcus Clarke and Ion-Mihai Tetcu stepped down from their duties. Joe was our longest serving member of the team. Among his many accomplishments was being the repocopy source of authority, instrumental in championing tinderbox development and maintaining portlint. Ion-Mihai is the person who initially gave us Ports QAT, which provided almost immediate response to a problematic commit -- and then he enhanced with other tests such as installation of all ports to a non-standard DESTDIR. These combined efforts paved the way for cleaner commits to the ports tree.

VALUE-ADDED SERVICES FROM PORTMGR@

One of the behind-the-scenes jobs for portmgr@ is something called portmgr-feedback@. This was the original work of Port Manager Emeritus Mark Linimon and was a way to combine his responsibilities on portmgr@ and bugmeister@. When ports are old, stale, unfetchable, or in some state of disrepair, the maintainer is contacted with a cc: to portmgr-feedback@. Any replies that come from the maintainer are the responsibility of various members of portmgr@ to try to resolve. Most responses are last-ditch efforts of contributors who just want their port updated. In recent months, members of the portmgr-lurkers have earned their stripes working in this capacity.

TIPS FOR PROSPECTIVE PORTERS

Staging all ports is the future of the ports tree. In the old model of the tree, you would compile your port in a scratch folder and then install it to its destination as root. If the port had been improperly prepared or if something went wrong during the install phase, you might have remnants of an incomplete installation. With staging, your port builds in your scratch directory, and then installs into your staged directory, generating a tarball upon completion. It is from this tarball that your port gets installed to its final destination.

The first step is to remove the NO_STAGEDIR=yes from your ports Makefile. The next steps vary based on the build system of your port source code. Sometimes it may just be a matter of changing some build targets in your Makefile, other times you might have to customize an install phase. This is where you, as a porter, have to use your imagination. The great thing is that a lot of people have already staged their ports and you already have some good examples in the tree.

One of the monotonous tasks of maintaining ports is the plist generation. If you have your port staged correctly, you can simply issue the command "make makeplist" and the infrastructure will generate the list to stdout, you can redirect this output to pkg-plist and your work is almost done. (You have done full testing on your port right?:-D)

If your port relies on cmake to build, you are more fortunate than you can imagine. You do not have to imbed any additional directives in your Makefile for staging, the infrastructure just takes care of it for you, and you just have to capture your plist.

You can read more about staging at https://wiki.freebsd.org/ports/StageDir

DOING YOUR PART TO IMPROVE THE PORTS TREE

At any given time, approximately 20% to 25% of ALL ports are unmaintained. Orphaned ports need updates too! If you have one installed on your system, consider adopting it. The following shell script will generate a list of ports you have installed on your system that do not have a maintainer cd /usr/ports; grep -F "`for o in \`pkg_info -qao\` ; \ do echo "|/usr/ports/${o}|" ; done`" `make -V INDEXFILE` | \ grep -i \| ports@freebsd.org\| | cut -f 2 -d \|

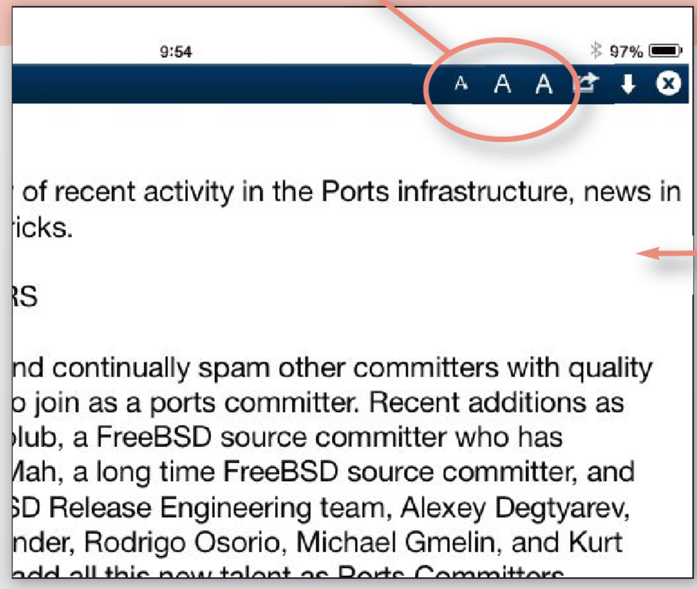
Once you have identified some prospective ports, do your research to see if the port needs updating and file a Problem Report to update and claim ownership of the port.

You can read more about adopting ports at, http://www.freebsd.org/doc/en/articles/contributing-ports/adopt-port.html.

The ports tree offers you a lot of software that gives you advanced usability, but it does not happen by itself. It is a user driven effort, so get involved in some way, whether it is adopting a port, or offering patches for other ports.

http://fb.me/portmgr -- "Like" us
http://twitter.com/freebsd_portmgr -- Follow us

3. **You can also select a type size.**



of recent activity in the Ports infrastructure, news in
ricks.

RS

nd continually spam other committers with quality
o join as a ports committer. Recent additions as
lub, a FreeBSD source committer who has
Mah, a long time FreeBSD source committer, and
SD Release Engineering team, Alexey Degtyarev,
nder, Rodrigo Osorio, Michael Gmelin, and Kurt
add all this new talent as Ports Committers

FreeBSD™ JOURNAL

Editorial Board

S&W PUBLISHING LLC
PO BOX 408, BELFAST, MAINE 04915

Publisher •	Walter Andrzejewski walter@FreeBSDjournal.com
Editor-at-Large •	James Maurer jmaurer@FreeBSDjournal.com
Art Director •	Dianne M. Kischitz dianne@FreeBSDjournal.com
Office Administrator •	Cindy DeBeck cindy@FreeBSDjournal.com
Advertising Sales •	Walter Andrzejewski walter@FreeBSDjournal.com Call 888/290-9469

FreeBSD Journal (ISBN: 978-0-615-88479-0) is published 6 times a year. (January/February, March/April, May/June, July/August, September/October, November/December)

Published by the FreeBSD Foundation,
PO Box 20247, Boulder, CO 80308
ph: 1/720-207-5142 • fax: 1/720-222-2350
e-mail: board@freebsdfoundation.org

FreeBSD Journal Does Networking and Gets Even More Dynamic!

Here we are with Issue 3, and at this juncture we're making available a Dynamic Edition (DE) in addition to our mobile offerings. We've had several requests from folks who wanted to read the Journal on their web browsers and the DE makes that possible. When we first began the Journal, we wanted to make sure to provide an experience not just equal to, but better than, paper magazines or plain PDF files. Our design work went into making the Journal app a clean, clear, interesting and instructive experience. With two issues out the door and being read on devices everywhere in the world, we turn our attention to what we can do for folks who really prefer to consume their content in a browser. We think we've gotten the right mix of features from the app well integrated into the browser version, the Dynamic Edition. A one-year subscription (6 issues) to the Dynamic Edition is $19.99, the same price as the mobile app. For more information on the Dynamic Edition, go to the home page of the FreeBSD Foundation Website (www.FreeBSDFoundation.org). As the Journal evolves, everyone will see new features appear in both the app and DE versions. Our intent is to keep these two delivery vehicles feature locked so that no matter which version you're using, you get the same, high-quality experience.

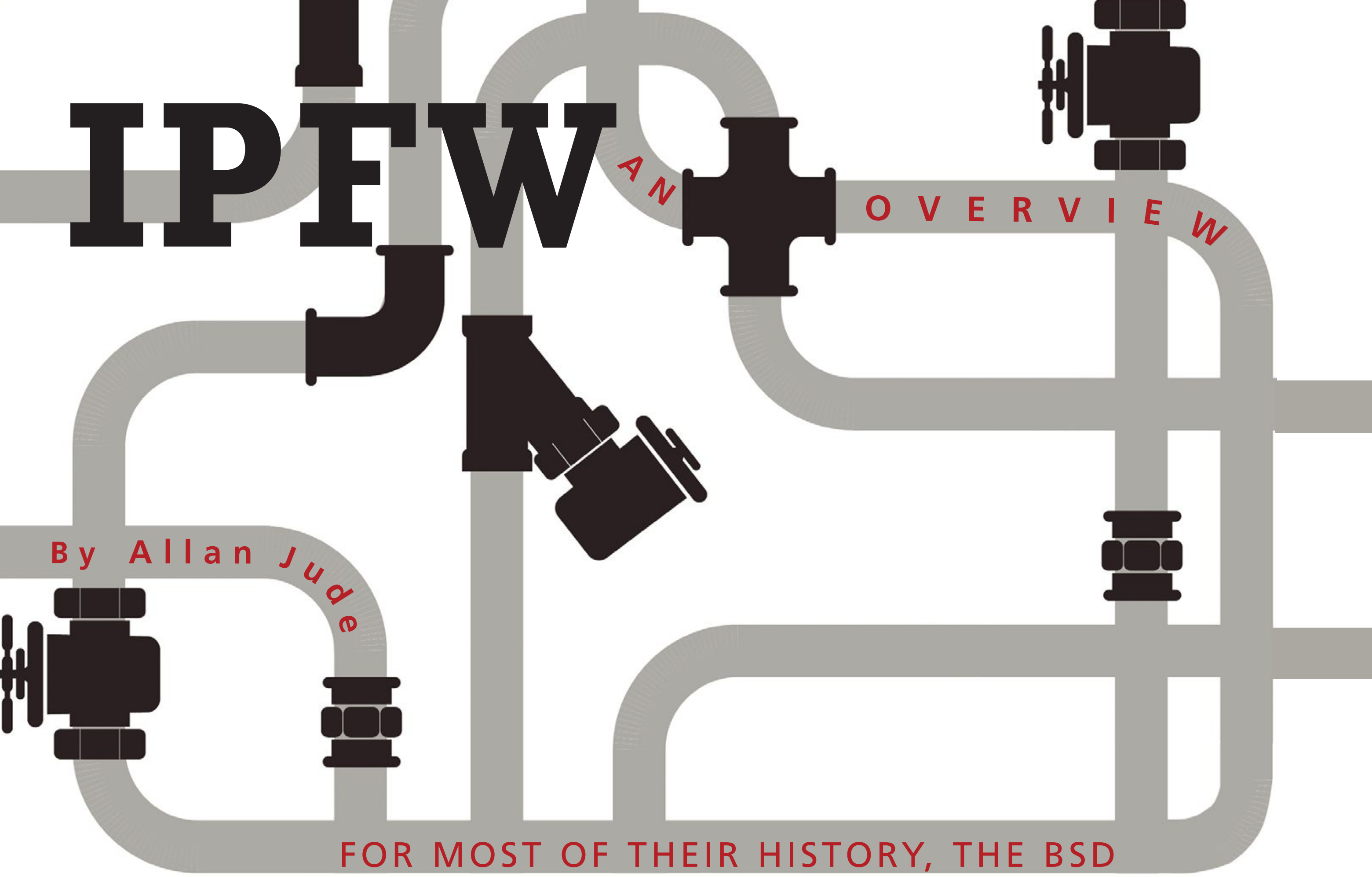
In this issue, our feature articles are all about networking. It was the 4.x series of BSD releases—from the Computing Systems Research Group at UC Berkeley—that first put the TCP/IP protocols and the sockets API into the operating system, which gave rise to the modern Internet. FreeBSD has carried on its tradition of excellence and innovation in networking, which is why it is used in so many diverse products, including storage systems from NetApp and EMC/Isilon as well as routers from Juniper, all of which depend on having superior network software.

Today FreeBSD has a large number of interesting networking features for systems programmers as well as systems administrators, and we cover just four of these in this issue. Mark Johnston writes about using the DTrace framework to look at networking activity in the system. DTrace, originally written for the Solaris operating system, offers some very unique abilities for looking deeply into system performance and applying it to the networking sub-systems, leading to a far better understanding of what's going on "under the hood." Randall Stewart takes us through an excellent tutorial on how to use kqueues, which are often employed for building high-performance networking programs, although the kqueue system itself can be used with any type of I/O on the system. Michael Lucas, well known author on topics as diverse as mastering SSH and FreeBSD in general, writes about securing the Domain Name System (DNS) with DNSSEC. DNS is a critical piece of Internet infrastructure that must be secured if we are to continue to have a working Internet. Allan Jude returns with a piece on IPFW, one of the two firewalls available in FreeBSD. Allan explains how to write firewall rules so that you get the highest performance from IPFW. Michael Bentkofsky and Julien Charbon, both from Verisign, discuss scaling performance with TCP. Both are in a unique position to see problems at scale because of the amount of connections that Verisign serves every day.

The Journal's columnists provide another round of great updates on the world of FreeBSD, with Glen Barber covering what's up in the source tree, Thomas Abthorpe on Ports Report, and Dru Lavigne continues to do double duty with Events Calendar and her This Month column on the history of FreeBSD.

As usual the editorial board is thrilled that so many people have subscribed and are reading the Journal. We welcome your comments, questions and feedback at: feedback@FreeBSDJournal.com.

Sincerely, **FreeBSD Journal Editorial Board**



IPFW An Overview

By Allan Jude

FOR MOST OF THEIR HISTORY, THE BSD family of operating systems has been known for making great firewalls. IPFW receives less attention than the PF packet filter, but it is very well featured with many advantages. IPFW was first introduced with FreeBSD 2.0 in 1994, while dummynet functionality came along in 2.2.8 (1998).

The current incarnation of IPFW, a complete rewrite dubbed IPFW2, was written and introduced in the summer of 2002. IPFW is remarkably fast and has very good SMP scalability.

IPFW is a "first match" firewall, meaning that each packet is compared against a numbered rules list, and once a rule matches, the search ends. This allows the administrator to write the rules in a specific order to achieve the greatest speed, and avoid comparing certain packets against more complex rules. Bandwidth and quality can be defined with pipes and queues, enforced with rules. IPFW also features an in-kernel NAT implementation which augments the existing user-space natd, full support for VIMAGE/VNET which creates a separate instance of the firewall in each VNET jail, multiple rule sets, dynamic rules, and tight integration with the operating system to provide features including rule matching against the user or jail which generated the packet.

This article covers the basics of enabling and configuring IPFW. It then discusses some advanced topics including rule numbering advice, simulating real world networks, traffic prioritization and shaping, and using the in-kernel NAT implementation, including configuring port forwarding, in conjunction with jails. The article then ends with an overview of some of the other features of IPFW.

LOADING THE FIREWALL

While IPFW can be compiled into a custom kernel, support is usually enabled by loading its kernel loadable module.   Since the default policy is to deny all traffic, one should also either create a custom rule set, or load a sample rule set. The built-in /etc/rc.firewall script contains the logic for a number of basic firewall rulesets. The available templates are: open, closed, simple, client, and workstation.

To enable IPFW, but not block any traffic, add the following lines to /etc/rc.conf:

```
firewall_enable="YES"
firewall_type="OPEN"
```

To create a simple stateful firewall:

```
firewall_enable="YES"
firewall_type="CLIENT"
firewall_client_net="192.168.0.0/24"  #Use the IP for your internal network
firewall_client_net_ipv6=""  #Specify the internal IPv6 net if you have one
```

Another option is to indicate the full path of a custom rule set file as the firewall_type, and IPFW will read that file and interpret each line as the arguments of the IPFW command. IPFW also supports using a preprocessor (such as m4) on the specified file, indicated by the -p flag. This allows an administrator to create a single template that, when preprocessed, generates the firewall rules specific to each different host.

Once the rules are in place, "service ipfw start" will start the firewall and apply the rules. Remember to nohup the command if working remotely, otherwise the connection may be closed when the firewall module is loaded, before the rules that allow the connection are added.

TIP: The /usr/share/examples/ipfw change_rules.sh script will apply a new set of firewall rules, and then prompt the administrator to confirm that they are satisfied with the new rules. If the administrator is not satisfied, or is locked out by the new rules, the old rules are restored after 30 seconds.

SIMPLE RULES

IPFW rules are fairly straightforward and easy to write. The following is a brief walkthrough of the basic administration commands for the OPEN firewall:

To show existing rules use 'ipfw list':

```
# ipfw list
00100  5084973634  3455207967775 allow ip from any to any via lo0
00200           0              0 deny ip from any to 127.0.0.0/8
00300           0              0 deny ip from 127.0.0.0/8 to any
65000 94077116494 77774399268246 allow ip from any to any
65535           0              0 deny ip from any to any
```

The first column is the rule number. Each rule is assigned a number which determines the order in which packets are compared. It is possible to have multiple rules with the same number, but this makes it more difficult to manage those rules. The next field is the number of packets that have matched the rule, followed by the total number of bytes of those matching packets. These counters can be reset using the "zero" sub-command to "ipfw". The remainder of each line is the rule.

To add a basic rule which blocks incoming connections to port 25, use this command:

```
# ipfw add 5001 unreach port tcp from any to me dst-port 25
```

This creates rule number 5001. A rule is always created using the keyword "add", and takes effect immediately. In this rule, the action that will be applied to a matching packet is "unreach port", which will generate an ICMP reply informing the remote host that the port is not accessible, as opposed to "deny" which will silently drop the packet. The body of the rule "tcp from any to me dst-port 25" determines which packets match. The first keyword is the protocol (ip, icmp, tcp, udp, etc). The next section of the body indicates the source and destination addresses of the packet; it allows keywords "any" and "me", which will match any address assigned to any interface on the machine. Finally, additional options can be specified, including source and destination port (src-port and dst-port), direction (in or out), interface (via em0), attempts to create a new connection (keyword "setup", packets with the SYN flag set, but the ACK flag not set), previously established connections (keyword "established", packets with ACK or RST flags set), and most other IP and TCP protocol headers.

An example of a more advanced rule:

```
# ipfw add 5002 allow log logamount 50 ip from 192.168.0.0/24 to any dst-port 80 setup
```

This creates rule number 5002 (for consistency, and administrator sanity, example rules will be numbered up from 5000). This rule will match, log and allow attempts to establish a new connection from the specified subnet to any host on port 80. A log limit has been set to 50 entries: while additional packets will be allowed, only the first 50 packets that match this rule will be logged in order to prevent the logs from filling entries for allowed packets. If needed, the log counter can be reset to 0 with the "ipfw resetlog" command.

NAMING

Rules, pipes and queues each have a separate rule number space, ranging between 1 and 65535. This means it is possible to have a rule and a pipe with the same number, without them being related. Whatever numbering system works for you, keep in mind that having some kind of functional isolation in your numbering is beneficial if you are working on a firewall (inevitably remotely, and usually late). Isolating the numbering system (5000s for rules, 30000s for pipes, 40000s for queues) and potentially grouping into related numeric sets (5001, 30001, 40001), can assist with clarity and avoid confusion.

SIMULATING REAL-WORLD NETWORKS

It is often desirable to test an application under real world conditions, where the network is not a quiet LAN with no latency or congestion. IPFW has a feature to simulate the real world conditions of the public Internet called dummynet(4). Dummynet provides the ability to artificially limit, queue, delay or drop packets to create the desired simulated network conditions.

To enable dummynet functionality, the kernel module must be loaded. This can be done automatically by adding the following to /etc/rc.conf

```
dummynet_enable="YES"
```

The most basic example of dummynet(4) is creating a pipe with limited bandwidth:

```
# ipfw pipe 30003 config bw 5Mbit/s
# ipfw add 5003 pipe 30003 ip from any to 192.168.0.101
```

The first command configures pipe 30003 with a bandwidth of 5 megabits per second. Pipe numbers are separate from rule numbers (for consistency, and administrator sanity, example pipes will be numbered in the 30000s, with the least significant bits matching the corresponding rule). The second command creates firewall rule 5003 using the "add" keyword. Rule 5003 directs all matching traffic (that has not matched a previous rule) through pipe 30003. This will effectively limit the downstream bandwidth of the host 192.168.0.101 through the firewall to 5 megabits per second.

This is useful for shaping traffic from specific hosts, or simulating slower links, but it does not provide an especially real simulation in terms of qualitative network variability. A better simulation would look like this:

```
# ipfw pipe 30003 config bw 5Mbit/s delay 150 burst 128k plr 0.001 queue 32KBytes noerror
```

This will reconfigure pipe 30003 with convenient options that can help to reproduce application behavior caused by network issues. The delay option will add 150ms of latency to each packet, simulating the latency between Los Angeles and London. The burst option will allow slightly more than the maximum amount of bandwidth to be used, if the pipe was not full beforehand. If the pipe is idle then the first 128kb of data is passed without being rate limited. The next option simulates a packet loss rate (plr) of 0.1%, causing an occasional retransmit, as might be expected on a less than ideal network. The queue option sets the maximum amount of excess data (in packets or KBytes) that will be accepted before additional packets are refused. This can be used to simulate buffer bloat, by setting a low rate limit with a large queue. The final option, noerror, causes the firewall not to return an error to the calling application when a packet is dropped.
Normally, if more data is trying to be sent than can be transmitted within the rate limit set, the firewall notifies the calling application with the same error that would be returned if the device queue was full on an unrestricted network. By suppressing this error, it simulates loss at an upstream router further along the path, where the application will be unaware that the packet has been dropped until it is not acknowledged on the other side of the connection.

CONTROLLING THE FLOW OF DATA

Pipes can also be allocated dynamically. For example, if there are many clients behind the firewall, each client can be limited to a 5Mbit/s flow by creating a dynamic pipe based on a mask (/24 in this case):

```
# ipfw pipe 30004 config bw 5Mbit/s mask src-ip 0x000000ff
# ipfw add 5004 pipe 30004 ip from any to 192.168.0.0/24
```

It is also possible to mask based on destination ip, source or destination port, or protocol.

One other option is to mask 'all' bits (source and destination IP, source and destination port, and protocol). This limits any single connection (flow) to 5Mbit/s, but allows each client multiple connections at this speed:

```
# ipfw pipe 30004 config bw 5Mbit/s mask all
```

Sometimes the goal of traffic shaping is not to limit the traffic of any one host, but to ensure that a set amount of bandwidth is shared equally among a set of hosts. In this case, rather than pushing all of the bandwidth directly through a pipe, the firewall can be used to create a number of queues with different priorities in order to classify the different types of traffic.

When a dynamic queue is created with a mask, each flow (in the following case, 1 per source ip address in the subnet) shares a parent pipe evenly. Create a pipe with limited bandwidth and then create a queue to use that pipe. Queue numbers are separate from pipe numbers (for consistency, and administrator sanity, example queues will be numbered up from 40000). Add a rule that matches the desired traffic. The queue rule will create a dynamic queue for each unique flow identifier, as determined by the specified mask. Each flow will have equal access to the limited pipe.

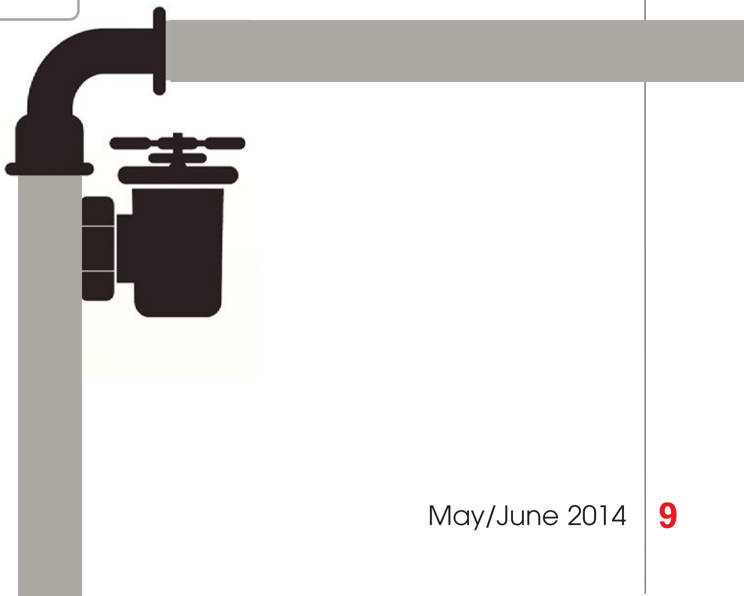
```
# ipfw pipe 30005 config bw 75Mbit/s
# ipfw queue 40005 pipe 30005 mask src-ip 0x000000ff
# ipfw add 5005 queue 40005 ip from any to 192.168.0.0/24
```

Contrast this to dynamic pipes, where each source ip address (flow identifier) had its own separate rate limit.

Sometimes, sharing equally is fine. However "all hosts are equal, but some hosts are more equal than others". Queues can be weighted, allowing certain traffic to get a greater share of the available pipe:

```
# ipfw pipe 30006 config bw 75Mbit/s
# ipfw queue 40006 pipe 30006 mask src-ip 0x000000ff weight 5
# ipfw queue 40007 pipe 30006 mask src-ip 0x000000ff weight 25
# ipfw add 5006 queue 40006 ip from any to 192.168.0.0/24
# ipfw add 5007 queue 40007 ip from any to 192.168.1.0/24
```

Creating a pipe to allocate bandwidth and two queues with different weightings, with supporting



subnet rules, means that hosts in the second subnet get higher priority access to the allocated bandwidth.

This same style of traffic management can also be applied to specific applications and services. Set up a pipe, add two differently weighted queues, plug the queues into rules for the specific service, and add a final catch all:

```
# ipfw pipe 30008 config bw 50Mbit/s queue 20
# ipfw queue 40008 pipe 30008 mask all weight 100
# ipfw queue 40011 pipe 30008 mask all weight 10
# ipfw add 5008 queue 40008 ip from any to any dst-port 5060
# ipfw add 5009 queue 40008 ip from any to any src-port 5060
# ipfw add 5010 queue 40008 ip from any to any iptos lowdelay
# ipfw add 5011 queue 40011 ip from any to any
```

The above rule set creates a pipe with 50 megabits per second of bandwidth and a maximum queue of 20 packets. Two queues are then created, where the first has a weight 10 times higher than the second. Traffic is then classified into one of these two queues. Packets with a source or destination port of 5060 (SIP), or packets with the IPTOS_LOWDELAY flag go into the high priority queue (40008), and the rest of the traffic goes into the low priority queue (40011). This should help ensure that VoIP calls do not suffer during periods of peak network activity.

Network traffic can also be shaped based on criteria specific to the machine the firewall is on. IPFW can match traffic based on the user, group, or jail that generated the traffic:

```
# ipfw pipe 30014 config bw 100Mbit/s
# ipfw pipe 30015 config bw 5Mbit/s
# ipfw pipe 30016 config bw 10Mbit/s
# ipfw add 5012 allow ip from any to any uid root
# ipfw add 5013 allow ip from any to any gid wheel
# ipfw add 5014 pipe 30014 ip from any to any jail 4 in
# ipfw add 5015 pipe 30015 ip from any to any jail 4 out
# ipfw add 5016 pipe 30016 ip from any to any
```

This set of rules will shape the traffic based on the user or jail that generated it. The first three commands configure pipes with specific amounts of available bandwidth. The next two rules allow all traffic generated by root, or members of the wheel group, to pass unshaped. The next pair of rules matches traffic flowing in and out of a specific jail, creating an asymmetric connection, limiting traffic to 100mbps inbound, but only 5mbps outbound. The final rule matches all other traffic (other users and jails) and limits them to 10mbps total (not per direction).

BASIC NAT FOR A JAIL

IPFW can be useful if you need to quickly setup basic NAT to allow a number of jails on a public facing machine to access the Internet, without each having a dedicated IP address. This example assumes the jails have internal IP addresses bound to lo0.

To enable NAT, add the following to /etc/rc.conf:

```
        gateway_enable="YES"
firewall_enable="YES"
firewall_type="OPEN"
firewall_nat_enable="YES"
firewall_nat_interface="em0"                    #public interface
firewall_nat_flags="redirect_port tcp 10.99.0.2:80 80 redirect_port tcp 10.99.0.2:443 443"
```

This will create an open firewall that will NAT outbound traffic via the IP address assigned to em0. It also configures port forwarding, to redirect inbound traffic on ports 80 and 443 to the private IP of the jail.

ADDITIONAL FUNCTIONALITY

IPFW has a number of other keywords that can be used to create advanced rule sets. The "prob" keyword, as part of the rule action, determines the probability that a packet will match the rule. Using this, the administrator can construct rules to direct portions of traffic in different ways, for split testing, load balancing, or simulating failure. Packets can also be "tagged" with numeric ID numbers to be used in later rules for things such as establishing trust relationships between interfaces. IPFW includes a forwarding capability; the "fwd" keyword will change the internal next-hop field of the packet as it passes through the firewall. This does not modify the headers of the packet, but changes how the kernel will route the packet and is ideal for IP- or port-based load balancing. IPFW can create a software monitor port with the "tee" keyword, which will send a copy of each matching packet to userland via a divert(4) socket. IPFW may also be used to mark packets with a specific FIB (Forwarding Information Base), causing matching packets to be routed using a specific kernel routing table.

CONCLUSION

This article only begins to scratch the surface of the capabilities and features of IPFW. The IPFW man page provides extensive documentation of each feature with plentiful examples. The FreeBSD handbook also includes a chapter on IPFW with additional explanation and examples. Users with questions are encouraged to address them to the freebsd-questions mailing list or post on the FreeBSD Forums. ●

Allan Jude is VP of operations at ScaleEngine Inc., a global HTTP and Video Streaming Content Distribution Network, where he makes extensive use of ZFS on FreeBSD. He is also the host of the video podcasts "BSD Now" (with Kris Moore) and "TechSNAP" on JupiterBroadcasting.com. Allan is currently working on earning his FreeBSD doc commit bit, improving the handbook and documenting ZFS. He taught FreeBSD and NetBSD at Mohawk College in Hamilton, Canada from 2007-2010 and has 12 years of BSD unix sysadmin experience.

BY MARK JOHNSTON

DTrace

FreeBSD offers a plethora of tools and tricks for answering questions about activity in the network stack.

Utilities like systat(1) and netstat(1) are great at giving us a few starting points: they can, for instance, show packet and bit rates on a per interface or per protocol basis. More sophisticated tools will make use of the Berkeley Packet Filter (BPF) to track individual packets as they enter and leave the system; iftop (available in ports as sysutils/iftop) uses this technique to display bit rates per 4-tuple. The venerable tcpdump(1) also uses the BPF interface to capture and log packets in real time, making it possible to answer questions based on post-capture analysis.

With FreeBSD 10, the kernel contains a new set of DTrace probes which give users a great deal of visibility into the inner workings of the network stack. Specifically, users can now write scripts based on packet send and receive events in the IP, TCP and UDP layers of the FreeBSD kernel, and additionally peer into the internal TCP state of a given connection in real time. This is a powerful addition to any programmer's or sysadmin's toolbox, since it provides a framework for answering arbitrary questions about the behavior of FreeBSD's IP stack; rather than being limited by the output of existing utilities, DTrace allows people to develop their own tools to explore network activity, whether the motivation is to monitor performance indicators, pinpoint the source of problems, or to simply learn more about network protocols.

This article will give an overview of each of the new probes, explaining their use and providing examples. It assumes a basic knowledge of and familiarity with DTrace, but all of the examples used this article are either dtrace(1) commands—which can be run directly from the shell—or executable scripts. They have all been developed and tested on FreeBSD 10, and readers with an available testing system are encouraged to try and run them to get a feel for DTrace's capabilities. They are available for download at http://people.freebsd.org/~markj/dtrace/network-providers/examples/.

One caveat is that the FreeBSD implementation of these probes is relatively new, so it is naturally possible that you may run into bugs or hard-to-explain behavior when experimenting with them. DTrace guarantees that scripts cannot crash the system or otherwise corrupt its state, so there is no danger in running the examples or any DTrace scripts on FreeBSD. However, if you run into problems making use of the new probes or when using DTrace, please report them on the freebsd-dtrace@FreeBSD.org mailing list. Problems and questions related to DTrace which aren't specific to FreeBSD should be reported on the dtracediscuss@lists.dtrace.org mailing list; many of the original and current developers of DTrace are subscribed and will readily respond to posts on this list.

DTrace on FreeBSD

Users of FreeBSD's DTrace implementation will possibly be familiar with the fbt provider, which is used to trace function calls in the FreeBSD kernel as they occur. This provider is exceptionally handy for users already familiar with FreeBSD kernel internals, but suffers from a few downsides:

- Its use requires a moderately good understanding of kernel code, which is an unreasonably high barrier to entry for most users wishing to write their own scripts.
- The fbt probes are by definition tightly coupled to kernel code; if the code underlying a script changes, the script may fail to run or may produce incorrect results. So scripts written for one version of FreeBSD may not work on another, and almost certainly won't work on other operating systems.
- Individual fbt probes often do not correspond nicely to logical system events. Suppose you wish to write a DTrace script which prints the destination address of each IP packets as FreeBSD hands them over to the network card driver. It turns out that this is a discouragingly difficult task: it involves instrumenting at least four different functions in various parts of the IPv4 and IPv6 code, each of which is called with different arguments.

The new network probes allow users to write scripts and trace network-related events using an interface that doesn't suffer from the problems above. They provide a stable and simple window into the kernel's activity: to trace the destination address of IP packets, simply run:

Example 1:

```
# dtrace -n 'ip:::send {printf("%s", args[2]->ip_daddr);}'
```

Implementing this exact functionality using the fbt provider would involve writing at least fifty lines of fairly impenetrable D code, at least by the author's estimation after a half-hearted and mostly failed attempt at the exercise. By comparison, this example is quite straightforward and transparent, aside from the somewhat enigmatic "`args[2]`." In English, it essentially reads as "every time we send an IP packet, print its destination address." When this command is run on the author's laptop for several seconds while simultaneously pinging an internet address, we get the following output:

```
# dtrace -n 'ip:::send {printf("%s", args[2]->ip_daddr);}'
dtrace: description 'ip:::send ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0  36564                           :send 8.8.178.110
  0  36564                           :send 8.8.178.110
  0  36564                           :send 8.8.178.110
  0  36564                           :send 8.8.178.110
```

This is DTrace’s default output formatting. We can get more control by adding “`-x quiet`” to the dtrace(1) arguments and printing the newline (“\n”) ourselves:

```
    # dtrace -x quiet -n ‘ip:::send {printf(“%s\n”, args[2]->ip_daddr);}’
    8.8.178.110
    8.8.178.110
    8.8.178.110
    8.8.178.110
```

Within a D script, the same effect can be achieved by adding the following line to the beginning of the file:

```
#pragma D option quiet
```

The New Network Probes

The network probes discussed in this article originate from Solaris and are also present in illumos and OS X. They belong to the new ip, tcp and udp DTrace providers; the following command in Example 2 will list them on your system:

Example 2:

```
# dtrace -l -P ip -P tcp -P udp
ID          PROVIDER     MODULE                   FUNCTION NAME
36492       ip           kernel                   receive
36493       ip           kernel                   send
36494       tcp          kernel                   accept-established
36495       tcp          kernel                   accept-refused
36496       tcp          kernel                   connect-established
36497       tcp          kernel                   connect-refused
36498       tcp          kernel                   connect-request
36499       tcp          kernel                   receive
36500       tcp          kernel                   send
36501       tcp          kernel                   state-change
36502       udp          kernel                   receive
36503       udp          kernel                   send
```

From this we can see that FreeBSD now has probes for IP, TCP and UDP packet send and receive events. The IP send and receive probes (i.e. `ip:::send` and `ip:::receive`) fire whenever FreeBSD sends or receives an IPv4 or IPv6 packet, respectively. Similarly, the TCP and UDP send and receive probes fire when the kernel sends or receives a TCP or UDP packet. As we can see, the tcp provider contains additional probes corresponding to TCP protocol events; for now, we will look at the send and receive probes and construct several examples.

Each of the network probes takes several arguments which together describe the packet which caused the probe to fire. The arguments themselves are each collections of related information. For instance, the third argument to `ip:::send` (“`args[2]`” in our earlier example) contains the high-level IP fields common to both IPv4 and IPv6: the IP version (`ip_ver`), the length of the payload (`ip_plength`), and the source and destination addresses (`ip_saddr` and `ip_daddr`). The fourth argument contains information describing the network interface used to transmit the packet, and the fifth and sixth arguments respectively yield the detailed IPv4 and IPv6 fields of the packet. That is, if the packet uses IPv4, “`args[4]`” will expose the fields of its IPv4 header, whereas an IPv6 packet will have its header fields exposed through “`args[5]`”. A comprehensive list and descriptions of the `ip:::send` probe arguments are available in [1], so they will not be duplicated here; corresponding pages are available for the tcp and udp providers at [2] and [3] respectively. Using the send and receive probes, we can print output to the terminal for each packet in real time. On a heavily loaded system, this will of course be impractical; in this situation, users will want to make use of DTrace’s data aggregation facilities or add predicates to allow selective tracing of packets. The following script prints basic information about each TCP packet as it enters and leaves the system. Note that forwarded TCP packets will not cause the tcp probes to fire since they are not examined by the TCP code.

Example 3 makes use of three probes. The `dtrace:::BEGIN` probe is used to print column

Example 3:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10Hz

dtrace:::BEGIN
{
    printf(“ %30s %-6s %30s %-6s %-6s %s\n\n”, “SADDR”, “SPORT”,
        “DADDR”, “DPORT”, “BYTES”, “FLAGS”);
}

tcp:::receive,
tcp:::send
{
    printf(“ %30s %-6u %30s %-6u %-6u (%s%s%s%s%s%s\b)\n”,
        args[2]->ip_saddr, args[4]->tcp_sport,
        args[2]->ip_daddr, args[4]->tcp_dport,
        args[2]->ip_plength - args[4]->tcp_offset,
        (args[4]->tcp_flags & TH_FIN) ? “FIN|” : “”,
        (args[4]->tcp_flags & TH_SYN) ? “SYN|” : “”,
        (args[4]->tcp_flags & TH_RST) ? “RST|” : “”,
        (args[4]->tcp_flags & TH_PUSH) ? “PSH|” : “”,
        (args[4]->tcp_flags & TH_ACK) ? “ACK|” : “”,
        (args[4]->tcp_flags & TH_URG) ? “URG|” : “”);
}
```

headers for the output of the tcp probe actions. As the headers suggest, the tcp probes print the 4-tuple associated with each TCP packet, along with the TCP payload size and the TCP flags. Some sample output shows NAT'ed SSH, HTTP and IMAPS traffic:

```
                      SADDR SPORT                           DADDR DPORT  BYTES  FLAGS

fe80:3::fa1a:67ff:fe03:f659 22         fe80:3::250:b6ff:fe0e:a825 42705  36     (PSH|ACK)
fe80:3::fa1a:67ff:fe03:f659 22         fe80:3::250:b6ff:fe0e:a825 42705  628    (PSH|ACK)
 fe80:3::250:b6ff:fe0e:a825 42705     fe80:3::fa1a:67ff:fe03:f659 22     0      (ACK)
fe80:3::fa1a:67ff:fe03:f659 22         fe80:3::250:b6ff:fe0e:a825 42705  100    (PSH|ACK)
 fe80:3::250:b6ff:fe0e:a825 42705     fe80:3::fa1a:67ff:fe03:f659 22     0      (ACK)
               192.168.0.27 41116                  173.194.76.108 993    37     (PSH|ACK)
             173.194.76.108 993                      192.168.0.27 41116  20     (ACK)
             173.194.76.108 993                      192.168.0.27 41116  63     (PSH|ACK)
               192.168.0.27 41116                  173.194.76.108 993    0      (ACK)
            173.252.102.241 443                      192.168.0.27 50220  429    (PSH|ACK)
               192.168.0.27 50220                 173.252.102.241 443    911    (PSH|ACK)
            173.252.102.241 443                      192.168.0.27 50220  20     (ACK)
               192.168.0.27 16039                    31.13.69.160 443    37     (PSH|ACK)
               31.13.69.160 443                      192.168.0.27 16039  20     (ACK)
```

The tcp probe action is a single printf() call which uses the third and fifth arguments to `tcp:::send` and `tcp:::receive`; these arguments contain the IP and TCP headers of the corresponding packet and make it easy to retrieve the 4-tuple associated with the packet. The TCP payload size is a bit trickier: the IP header contains the IP payload size, and the TCP header contains the offset from the beginning of the TCP header to the TCP payload; thus their difference gives the size of the TCP payload. Note that the TCP payload sizes reported by DTrace for outbound segments may be larger than the MSS for the connection if TSO is enabled on the outbound interface. Finally, `args[4]->tcp_flags` contains the segment's TCP flags, and the DTrace TCP library (found in `/usr/lib/dtrace/tcp.d` on FreeBSD) contains symbolic names for the each of the TCP flags.

Though the script above nicely demonstrates the information available through network probes, it is not particularly useful except as a learning tool since it generally prints an unmanageable amount of output on any system with continuous TCP activity. Fortunately, DTrace makes it easy to visualize aggregations of data, and the network providers can be used to examine the distributions of variables such as connection duration, latency, bitrates, packet count, and packet size. Moreover, DTrace's flexibility makes it possible to measure these quantities over virtually any independent variable(s): by host, port, L3 protocol, or network interface. It is also possible to examine network traffic on a per-process basis, though on FreeBSD this currently requires a special trick which will be shown later in the article.

A quick demonstration of this is given in Example 4, which will print a per-interface his-

Example 4:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10Hz

ip:::send,
ip:::receive
{
    @num[args[3]->if_name] = lquantize(args[2]->ip_plength, 0, 1500, 100);
}
```

Example 5:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10Hz

tcp:::state-change
/(args[5]->tcps_state == TCPS_CLOSED && args[3]->tcps_state == TCPS_SYN_SENT) ||
 (args[5]->tcps_state == TCPS_LISTEN && args[3]->tcps_state == TCPS_SYN_RECEIVED)/
{
    dur[args[1]->cs_cid] = timestamp;
}

tcp:::state-change
/(args[3]->tcps_state == TCPS_CLOSING ||
  args[3]->tcps_state == TCPS_FIN_WAIT_2 ||
  args[3]->tcps_state == TCPS_LAST_ACK) &&
 dur[args[1]->cs_cid] != 0/
{
   @["Connection duration (ms)"] = quantize((timestamp - dur[args[1]->cs_cid]) / 1000000);
    dur[args[1]->cs_cid] = 0;
}
```

togram of IP payload sizes using a linear distribution. This script does not print anything to the terminal while it is running; rather, it continuously collects data and prints a summary when it exits, which can happen when the user enters Ctrl-C or the script calls the built-in exit() function. In this case, the script runs until the user ends it:

In this case, a sample output on a system with a single active interface (wlan0 in this case) is

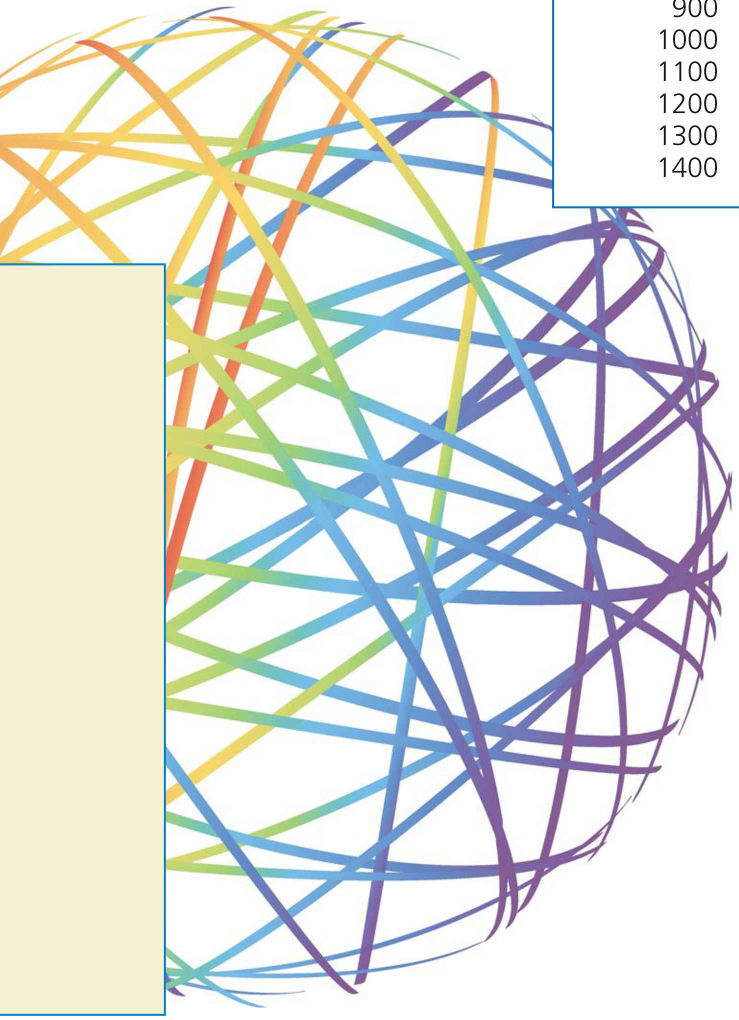
A system with multiple interfaces will print a histogram for each interface.

For another example, we can use the `tcp:::state-change` probe in Example 5 to show the distribution of TCP connection durations. This probe gives us the to and from states when the transitions happen, so we may measure durations by recording a timestamp when either

- a connection transitions from CLOSED to SYN-SENT, or
- a connection transitions from LISTEN to SYN-RECEIVED.

We then consider a connection to have ended once it enters the FIN-WAIT2, CLOSING or LAST-ACK states, once this happens, we may record the time difference between the two events. To store the initial connection timestamps, we use an array indexed by `args[1]->cs_cid`, an opaque integer which uniquely identifies a connection. That is, we can assume that multiple simultaneous connections will not share a connection ID. The full script then looks like this:

```
wlan0
         value------------------ Distribution ------------------- count
           < 0                                                     0
             0   |@@@@@@@@@@@@@@@@@@@@@@@@@                        479
           100   |@@@                                              61
           200   |@                                                16
           300   |@                                                24
           400   |@                                                27
           500   |@                                                14
           600   |                                                 5
           700   |                                                 10
           800   |                                                 5
           900   |                                                 9
          1000   |                                                 5
          1100   |                                                 5
          1200   |@                                                11
          1300   |@@@@@@@                                          152
          1400   |                                                 0
```



In the `tcp:::state-change` probe `args[5]->tcps_state` gives the from-state, and `args[3]->tcps_state` yields the to-state. Note that the second probe checks whether a timestamp for the connection exists by verifying that `dur[args[1]->cs_cid]` is non-zero: this is to ensure that we do not record data for connections that already exist when the script is started.

The examples presented thus far have hopefully convinced you of the potential of the network providers as building blocks for network tools, whether they are tailored to investigate some specific problem, or to track data that is difficult to obtain using existing monitoring tools. However, most of the examples we have seen so far could in principle be re-implemented with custom programs that use the BPF to intercept and inspect packets. On the other hand, the TCP probes give us a window into the internal state of the associated TCP connection; this is not accessible via BPF. This allows us to, for instance, monitor the TCP state transitions of connections as they occur:

Example 6

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10Hz

dtrace:::BEGIN
{
    printf(" %30s %-6s %30s %-6s %-9s\n", "LADDR", "LPORT",
        "RADDR", "RPORT", "DELTA(us)");
}

int last[uint64_t];

tcp:::state-change
{
    this->delta = last[args[1]->cs_cid] != 0 ?
        (timestamp - last[args[1]->cs_cid]) / 1000 : 0;
    last[args[1]->cs_cid] = timestamp;

    printf(" %30s %-6u %30s %-6u %-9u %s -> %s\n",
        args[3]->tcps_laddr, args[3]->tcps_lport,
        args[3]->tcps_raddr, args[3]->tcps_rport,
        this->delta,
        tcp_state_string[args[5]->tcps_state],
        tcp_state_string[args[3]->tcps_state]);
}

tcp:::state-change
/args[3]->tcps_state == TCPS_CLOSING ||
 args[3]->tcps_state == TCPS_FIN_WAIT_2 ||
 args[3]->tcps_state == TCPS_LAST_ACK/
{
    last[args[1]->cs_cid] = 0;
}
```

The script in Example 6 records a per-connection timestamp of the last state change in the “last” array, indexed by the connection ID stored in **args[1]->cs_cid**. Each time a state transition occurs, the 4-tuple associated with the connection is printed, along with the amount of time elapsed since the last state transition and the transition itself, e.g. “**state-established -> state-close-wait**”.

The **tcp_state_string** array is defined in **/usr/lib/dtrace/tcp.d** and provides string representations of each of the TCP states. Symbolic names for the states are also available, e.g. **TCPS_TIME_WAIT, TCPS_LAST_ACK**.

Note that we clear entries in the “last” array by setting them to 0 when TCP connections end, i.e. when they enter the FIN-WAIT-2, CLOSING or LAST-ACK states. At this point the connection state inside the kernel is about to be torn down, so the array entry will not be valid if the CID is reused for a future connection.

Advanced Tricks

Earlier we mentioned that it was possible to track network activity per process. In the future it will hopefully become possible to access the PID of the associated process through the network probe arguments, but this is currently not possible because of the way that the relevant data is organized within the kernel. However, using the fbt provider it is possible to associate PIDs or process names with the connection IDs available through **args[1]->cs_cid** in the TCP and UDP probes. The D code to accomplish this is somewhat opaque; however, it is included in the example below and can be reused in other scripts.

The following script in Example 7 prints a summary of per-process TCP activity every two seconds, reporting the total number of bytes transmitted and received over TCP, as well breakdown by process and 4-tuple:

This script is built up of several pieces. The `dtrace:::BEGIN` probe is used to initialize a pair of global variables used to count the number of TCP payload bytes that were transmitted and received in the current interval; they are reset by the `profile:::tick-2sec` probe, which prints summary data to the terminal every two seconds. The four FBT probes are used to populate and clear the `procs` array, which maps connection IDs to process names (e.g. “firefox” or “sshd”). In particular, the `fbt::tcp_usr_attach` probes fire when a new TCP socket is created, the

Example 7:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
    in = 0;
    out = 0;
}

fbt::tcp_usr_attach:entry
{
    self->so = args[0];
}

fbt::tcp_usr_attach:return
/args[1] == 0 && self->so != NULL/
{
    procs[(uintptr_t)self->so->so_pcb] = execname;
    self->so = NULL;
}

fbt::sosend_generic:entry, fbt::soreceive:entry
/args[0]->so_proto->pr_protocol == IPPROTO_TCP/
{
    procs[(uintptr_t)args[0]->so_pcb] = execname;
}

fbt::in_pcbdetach:entry
{
    procs[(uintptr_t)args[0]] = 0;
}

tcp:::send
/procs[args[1]->cs_cid] != “”/
{
    this->bytes = args[2]->ip_plength - args[4]->tcp_offset;

    out += this->bytes;
    @bytes[procs[args[1]->cs_cid], args[2]->ip_saddr, args[4]->tcp_sport,
        args[2]->ip_daddr, args[4]->tcp_dport] = sum(this->bytes);
}

tcp:::receive
/procs[args[1]->cs_cid] != “”/
{
    this->bytes = args[2]->ip_plength - args[4]->tcp_offset;

    in += this->bytes;
    @bytes[procs[args[1]->cs_cid], args[2]->ip_daddr, args[4]->tcp_dport,
        args[2]->ip_saddr, args[4]->tcp_sport] = sum(this->bytes);
}

profile:::tick-2sec
{
    out /= 1024;
    in /= 1024;
```

(Example 7 continues next page)

(Example 7 continued)

```
        printf(“%Y, TCP in: %6dKB, TCP out: %6dKB, TCP total: %6dKB\n”, walltimestamp,
             in, out, in + out);

        printf(“%-12s %-15s %5s %-15s %5s %9s\n”, “PROC”, “LADDR”, “LPORT”,
             “RADDR”, “RPORT”, “SIZE”);

        printa(“%-12s %-15s %5d %-15s %5d %@9d\n”, @bytes);
        printf(“\n”);

        trunc(@bytes);
        in = 0;
        out = 0;
}
```

`fbt::sosend_generic` and `fbt::soreceive` probes fire when a process transmits or receives data over TCP, and the `fbt::in_pcbdetach` probe fires when a TCP connection is closed.

The remainder of the script performs the actual per-process accounting. Each time the TCP stack sends or receives a packet corresponding to an entry in the `procs` array, the `in` and `out` global variables are incremented appropriately, and the bytes array is updated. This array is indexed by process name and the 4-tuple of the connection, and its contents are printed and cleared in the `profile:::tick-2sec` probe. This makes it possible to drill down into the TCP usage of specific processes, a task which is quite difficult without DTrace.

Of course, the script in Example 7 can be modified to perform different types of per-process or per-user accounting. For instance, one could keep running totals rather than clearing statistics every two seconds. When a process exits (signaled by the `proc:::exit` probe) its total TCP usage could be saved for later analysis. Additionally, with a few tweaks, Example 7 can be modified to track UDP usage rather than TCP, as shown in Example 8.

As a final example, we demonstrate how DTrace may be used to dig into some of the more advanced aspects of the TCP protocol.

Example 8:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
     in = 0;
     out = 0;
}

fbt::udp_attach:entry
{
     self->so = args[0];
}

fbt::udp_attach:return
/args[1] == 0 && self->so != NULL/
{
     procs[(uintptr_t)self->so->so_pcb] = execname;
     self->so = NULL;
}

fbt::sosend_dgram:entry, fbt::soreceive:entry
/args[0]->so_proto->pr_protocol == IPPROTO_UDP/
```

(Example 8 continues next page)

(Example 8 continued)

```
    {
        procs[(uintptr_t)args[0]->so_pcb] = execname;
    }

    fbt::in_pcbdetach:entry
    {
        procs[(uintptr_t)args[0]] = 0;
    }

    udp:::send
    /procs[args[1]->cs_cid] != “”/
    {
        /* Subtract UDP header size. */
        this->bytes = args[4]->udp_length - 8;

        out += this->bytes;
        @bytes[procs[args[1]->cs_cid], args[2]->ip_saddr, args[4]->udp_sport,
            args[2]->ip_daddr, args[4]->udp_dport] = sum(this->bytes);
    }

    udp:::receive
    /procs[args[1]->cs_cid] != “”/
    {
        /* Subtract UDP header size. */
        this->bytes = args[4]->udp_length - 8;

        in += this->bytes;
        @bytes[procs[args[1]->cs_cid], args[2]->ip_daddr, args[4]->udp_dport,
            args[2]->ip_saddr, args[4]->udp_sport] = sum(this->bytes);
    }

    profile:::tick-2sec
    {
        out /= 1024;
        in /= 1024;

        printf(“%Y, UDP in: %6dKB, UDP out: %6dKB, UDP total: %6dKB\n”,
                walltimestamp, in, out, in + out);

        printf(“%-12s %-15s %5s %-15s %5s %9s\n”,
            “PROC”, “LADDR”, “LPORT”, “RADDR”, “RPORT”, “SIZE”);

        printa(“%-12s %-15s %5d %-15s %5d %@9d\n”, @bytes);
        printf(“\n”);

        trunc(@bytes);
        in = 0;
        out = 0;
    }
```

Our main tool here is the fourth argument (`args[3]`) passed to each of the TCP probes. This argument contains information which describes the internal state of the connection, and is useful in exploring phenomena that are not easily observed by examining the headers in individual packets.

Here we use the TCP provider to detect the arrival of out-of-order segments; specifically, the script checks for inbound data packets whose sequence numbers do not match the next anticipated sequence number (accessed through `args[3]->tcps_rnxt`). If FreeBSD sees such a segment, it will add it to the connection's reassembly queue if there is space available; otherwise it is dropped. Out-of-

Example 9:

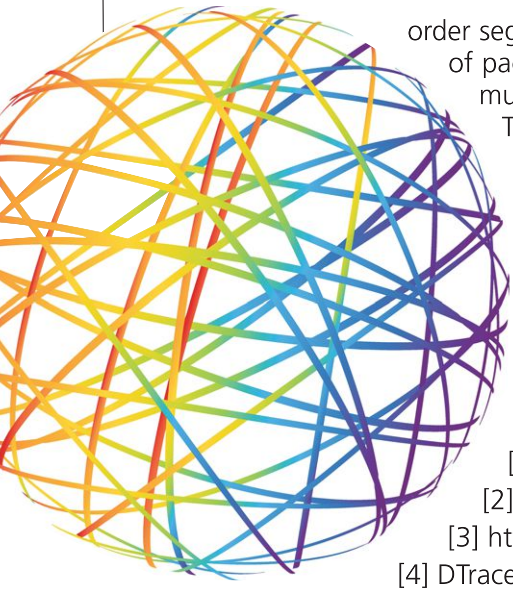
```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10Hz

tcp:::receive
/args[3]->tcps_state == TCPS_ESTABLISHED &&
 args[2]->ip_plength - args[4]->tcp_offset > 0 &&
 args[3]->tcps_rnxt != args[4]->tcp_seq/
{
   @invorderb[args[3]->tcps_raddr] = sum(args[2]->ip_plength - args[4]->tcp_offset);
   @invorderp[args[3]->tcps_raddr] = count();
}

tcp:::receive
/args[3]->tcps_state == TCPS_ESTABLISHED &&
 args[2]->ip_plength - args[4]->tcp_offset > 0/
{
   @valorderb[args[3]->tcps_raddr] = sum(args[2]->ip_plength - args[4]->tcp_offset);
   @valorderp[args[3]->tcps_raddr] = count();
}

dtrace:::END
{
   printf(“%-30s %-12s %-12s %-12s %-12s\n”, “RADDR”, “BYTES”,
        “OOO BYTES”, “PACKETS”, “OOO PACKETS”);
   printa(“%-30s %@-12d %@-12d %@-12d %@-12d\n”, @valorderb, @invorderb,
        @valorderp, @invorderp);
}
```



order segments may be the result of packet drops or of having multiple routes between the TCP endpoints; in general, they hurt throughput and should be investigated if they make up a large ratio of the total number of segments in a connection. The script in Example 9 counts out-of-order segments by remote host address. It also computes the total byte and packet counts for comparison. ●

Further Reading

The purpose of this article was to present an introduction to the network providers available in FreeBSD 10 and to give readers a feel for the sorts of problems that the providers are well-suited to address. A reference for the complete set of probes is available in [1-3], and they are also described in the excellent DTrace book by Brendan Gregg and Jim Mauro[4].

[1] https://wikis.oracle.com/display/DTrace/ip+Provider

[2] https://wikis.oracle.com/display/DTrace/tcp+Provider

[3] https://wikis.oracle.com/display/DTrace/udp+Provider

[4] DTrace: Dynamic Tracing in Oracle Solaris,Mac OS X, and FreeBSD.

[5] https://people.freebsd.org/~markj/dtrace/network-providers/examples/

Mark Johnston is a software engineer living in Waterloo, Ontario. He completed a bachelor's degree in mathematics at the University of Waterloo in 2013 and has been a FreeBSD user since 2010. He is interested in all aspects of operating systems development, with a particular emphasis on debugging and performance analysis utilities. Since obtaining a commit bit, his main focus has been on improving FreeBSD's DTrace implementation. He can be reached via email at markj@FreeBSD.org.

FOSSETCON

FREE AND OPEN SOURCE EXPO AND TECHNOLOGY CONFERENCE

FOSSETCON

2014

Come out and participate in the First Annual Fossetcon 2014 Florida's Only Free and Open Source Conference. With in 10 minutes of Disney Land, Universal Studios and Epcot Center.

BSD *Friendly*

DAY 0	FOOD, TRAINING, WORKSHOPS AND CERTIFICATIONS
DAY 1	FOOD, KEYNOTES, EXPO HALL, SPEAKER TRACKS
DAY 2	FOOD, KEYNOTES, EXPO HALL, SPEAKER TRACKS

FREE FOOD, TRAINING, CERTIFICATIONS AND GIVEAWAYS!!!

SEPT 11 - SEPT 13

ROSEN PLAZA HOTEL ORLANDO, FL

More info at
www.fossetcon.org

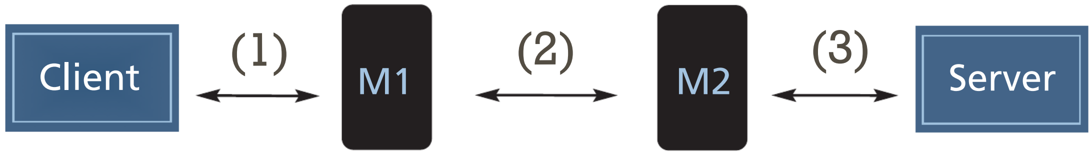
Fossetcon 2014: The Gateway To The Open Source Community

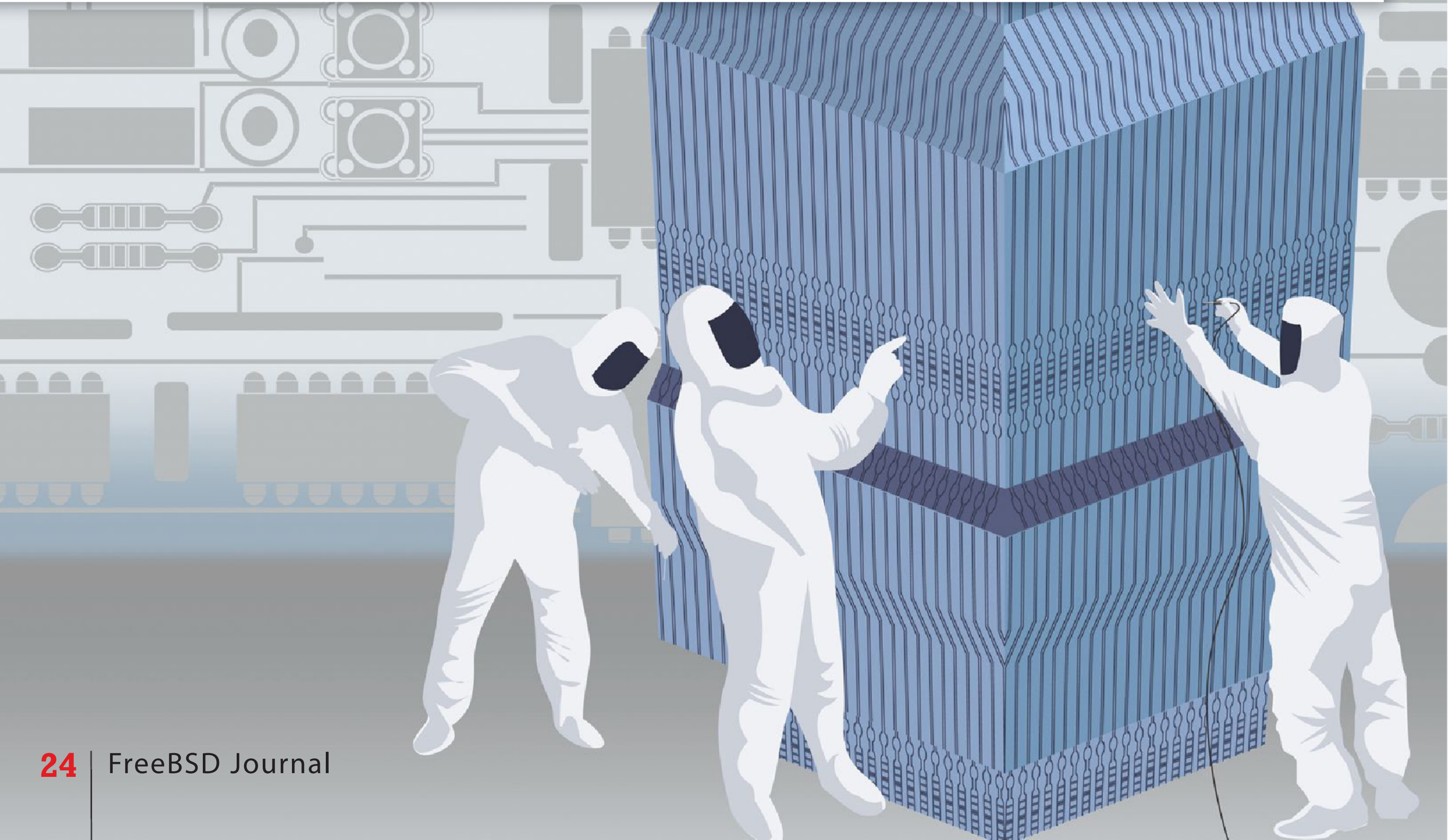
Kqueue madness

by Randall Stewart

Some time ago I was asked to participate in the creation of a Performance Enhancing Proxy (PEP) for TCP. The concept behind a PEP is to split a TCP connection into three separate connections. The first connection (1) is the normal TCP connection that goes from the client towards the server (the client is usually unaware that its connection is not going to the end server). The next connection (2) goes between two middle boxes (M1 and M2), the first middle box (M1) terminates the connection of the client pretending to be the server and uses a different connection to talk to the tail middle box (M2). This middle connection provides the "enhanced" service to the end-to-end connection. The final connection (3) goes between the tail middle box (M2) and the actual server. The figure below shows a diagram of such a connection.

A connection through a PEP





Now, as you can imagine, if you have a very busy PEP you could end up with thousands of TCP connections being managed by M1 and M2. In such an environment using poll(2) or select(2) comes with an extreme penalty. Each time a I/O event completes, every one of those thousands of connections would need to be looked at to see if an event occurred on them, and then the appropriate structure would need to be reset to look for an event next time. A process using such a setup can find itself quickly spending most of its time looking at and fiddling with the data structures involved in polling or selecting and has very little time for anything else (like doing the real work of handling the connections).

I arrived on this project late, but the team that was working on it had chosen to use a kqueue(2) instead of select(2) or poll(2). This, of course, was a very wise choice since they wanted to optimize the process to handle thousands of connections. The team used multiple kqueues and multiple threads to accomplish their end goal, but this in itself was another problem (when it came time to debug the process) since each kqueue had multiple threads and there were a **lot** of kqueues in the architecture. Depending on the number of cores in the hardware, the number of threads would increase or decrease (one thread for each kqueue for each core). This meant that in the first version of the software, a process might have over one hundred threads. After helping stabilize and ship the product I knew we had to improve things and this is where my dive into kqueue madness began.

In the re-architecture, I decided that I wanted to:

1. Keep the proxy untouched as far as the underlying work it was doing.
2. Reduce the number of threads to match the number of cores on the machine.
3. Insert under the proxy a framework that could later be reused for other purposes.

This led me to carefully craft a "dispatcher" framework that would help me accomplish all three goals.

In the middle of the rewrite, after I had gotten the framework pretty well complete and the proxy running on it, I began having issues with long-term stress testing. After deep debugging, I realized one of my problems was that I really did not fully understand the interactions that kqueues and sockets have. I found myself asking questions like:

a) What happens when I delete the kqueue event for a socket descriptor, yet do not close the socket?
b) Could I possibly see stale queued events that were yet to be read?
c) How does connect interact with kqueue?
d) What about listen?
e) What is the difference between all of the kqueue flags that I can add on to events and when do I use them properly?
f) What is the meaning of some of the error returns (EV_ERROR) and does that cover all my error cases?
g) Will I always get an end of file condition when the TCP connection closes gracefully directly from the kqueue?

So the first thing I did is access my friend Google and look for articles on kqueue (after of course reading the man page multiple times closely). Google found for me two articles, one by Jonathan Lemon [1] and the other by Ted Unangst [2]. Both articles are well worth reading but did not answer my questions. In fact the Ted Unangst article left me wondering if I did not use EV_CLEAR and failed to read all the data on a socket, I might miss ever seeing an event again, could this be the source of some of the problems I was seeing?

Since none of my questions were really answered, the only thing that made sense for me to do was write a special test application that I could use to test out various features of kqueues to answer my questions on our Operating System release based on FreeBSD 9.2. I decided to write this article so that you won't have to ponder these questions or write a kqueue test application. (You can find my test program at http://people.freebsd.org/~rrs/kqueue_test.c if you are interested.)

Kqueue Basics

In this section we will discuss the basic kqueue and kevent calls with all the "filters," "flags," and other goodies you need to know in order to transition your socket program into a kqueue based one. A lot of this information is in the man page kqueue(2) (a highly recommended source of information).

So the first thing you have to do is to actually create a kqueue. A kqueue is yet another file descriptor that you open (just like you do a socket) with a special call **kqueue().** There is nothing special about this call, just like creating a socket it returns a file descriptor index that you will use for all future calls. Unlike sockets it has no arguments whatsoever. It can fail, like any system call, and returns a negative 1, if it does fail, errno will be set to the reason. Once you have a descriptor returned from a successful kqueue call you use this with the kevent() system call to make all the magic happen.

The **kevent()** call has six arguments that it accepts which are:

- *Kq* – The actual kq that you created via the **kqueue()** call.
- *Changelist* – This is a pointer to an array of structures of type kevent that describes changes you are asking for in the kqueue.
- *Nchanges* – This is the number in the array of changes in the *Changelist*.
- *Eventlist* – This is another pointer to an array of structures of type kevent that can be filled in by the O/S to tell you what events have been triggered.
- *Nevents* – This is the bounding size of the *Eventlist* argument so the kernel knows how many events it can tell you about.
- *Timeout* – This is a struct timespec that represents a timeout value.

Like a **poll()** or **select()** the *Timeout* field allows you to control how "blocking" the call will be. If you set the argument to NULL, it will wait forever. If the *Timeout* field is non-NULL it is interpreted to be a timespec (tv_sec and tv_nsec) on the maximum time to delay before returning. One thing to note, if you specify zero in the *Nevents* field then the **kevent()** call will not delay even if a *Timeout* is specified.

Now the **kevent()** call can be used in one of three ways:

- As input to tell the kernel what events you are interested in (for example when you are setting up a bunch of socket descriptors, but you are not yet interested in handling the events). For this you would set *Changelist* to an array of at least one (possibly more) kevents and *Nchanges* to the number in the array. The *Eventlist* field would be set to NULL and the *Nevents* field would be set to 0.
- As only output, so you can find out what events have happened (for example when you are running in an event loop processing events, but possibly not setting any new ones in). To do this you would have the *Changelist* pointer set to NULL, the *Nchanges* set to 0, but *Eventlist* set to a pointer of an array of kevents you want filled and *Nevents* to the length of that array.
- The final way you can use it is to fill both sets, the in and the out, so that as you go in and wait for an event (or events) you can change what is being waited upon. This is useful in an event loop to minimize the number of kernel system calls you are making.

So that, in a nutshell, describes the calls and their use, but what is this kevent structure that you keep talking about? Well a kevent structure looks as follows:

```
struct kevent {
     uintptr_t ident;
     short filter;
     u_short flags;
     u_int fflags;
     intptr_t data;
     void *udata;
};
```

There is also a macro that is part of the event system that is a utility function that helps you setup this structure called **EV_SET()**. But first let's go through each field and describe what you put into it to get the results that you want:

- ***ident*** – This field is the identifier that you wish to have watched. In the case of a socket it would be the socket descriptor. For other kqueue calls it may be something else besides a descriptor. Each type of event to watch for specifies what the ident is made up of (generally it is some form of file descriptor, but in some cases its not e.g. a process id is used in one instance).
- ***filter*** – This field is the actual request that you are asking the kernel to watch for. The following list is the current filters that you can setup:

EVFILT_READ – A read filter looking for read events on a file or socket. This will be one of the filters we cover in a lot more detail.

EVFILT_WRITE – A write filter looking for when we can write to a file or socket descriptor. Again this will be one of the filter types that we take a closer look at.

EVFILT_AIO – asynchronous input output filter used in conjunction with the aio calls **(aio_read()/aio_write())**. For this article we will not discuss this kqueue event.

EFILT_VNODE – A file system change on a particular file. This is a very useful event (e.g. the **tail()** utility uses this when you are doing tail –f) but we won't be discussing this in this article.

EVFILT_PROC – A process event, such as a child's death or a fork of a child. Again a very useful event to watch for, especially if you are writing a process manager, but we won't cover it here.

EVFILT_SIGNAL – This event would come in when a signal occurs (after the signal arrives). This is again something for the reader to explore or maybe a future article ;-)

EVFILT_USER –A user generated event, which can be used to signal between threads or to wake a **kqueue()** for some other specific user defined reason (I use it for shutting down my framework actually). Again this is not something I will cover in more detail but I encourage the curious to investigate on their own.

- *flags* – This field, on input (I), tells the kernel what you want performed and on output (O) tells you what happened. The flags that are currently defined are:

EV_ADD(I) – Used when you wish to add your event (for us a socket descriptor) to the kqueue.

EV_DELETE(I) – Delete a previously added event from the kqueue.

EV_ENABLE(I) – Turn on a kqueue event. This may seem redundant but its not, since when you add an event, unless you specify this enable flag, it will not be watched for. You also use this after an event has triggered and you wish to re-enable it (if it does not automatically re-enable itself).

EV_DISABLE(I) – This is the opposite of enable, so you can send this down with a previously enabled filter and have the event disabled, but the internal kqueue structure inside the kernel will **not** be removed (thus its ready to be re-enabled with EV_ENABLE when you want).

EV_DISPATCH(I) – This flag tells the kernel that after it sends you an event, disable it until such time as you re-enable it.

EV_ONESHOT(I) – This flag tells the kernel that when the event occurs and the user retrieves the event, automatically delete (EV_DELETE) the kqueue entry from the kernel.

EV_CLEAR(I) – This toggles the state of the filter right after you retrieve the event so it "re-enables" itself automatically. Note that this can cause a "lot" of events to happen rapidly so this flag should be used very carefully.

EV_EOF(O) – This flag indicates the socket or descriptor has hit the end-of-file condition. For TCP this would be equivalent to reading 0 from the socket (i.e. the peer will send no more data and has sent a FIN).

EV_ERROR(O) – This flag indicates an error has occurred usually the data field will have more information. For example when a peer has "reset" the TCP connection with a RST, you will get an EV_ERROR with the data set to ECONNRESET or ECONNABORTED (or possibly some other errno).

- *fflags* – are filter specific flags on the way in and out. For example in the case of sockets you can use this as a way to change the SO_RCVLOWAT value (the *data* field would hold the new value). On the way out, if EV_ERROR were set, the *fflags* would hold the error just like our favorite error field errno.
- *data* – This field is used on a per filter basis to supply added information (e.g. the SO_RCVLOWAT mark).
- *udata* – This is a handy little pointer that is sent in to the kqueue call and will come up with the event. It is very useful to associate state with a particular event.

Now when sending kevents down to the kernel to be watched for, it's important to realize that to the kernel, the combination of a filter and ident make up a unique entry. So, for example, if I send down a EV_READ for socket descriptor 10 and a EV_WRITE for socket descriptor 10, these end up being two separate filters inside the kernel.

With these basics in mind one can write a simple event loop that would look something like:

```
int watch_for_reading(int fd)
{
        int kq, not_done, ret;
        struct kevent event;

        ret = -1;
        kq = kqueue();
        if (kq == -1) {
                return(ret);
        }
        not_done = 1;
        EV_SET(&event, EVFILT_READ, fd, EV_ADD|EV_ENABLE), 0, NULL);
        if (kevent(kq, &event, 1, NULL, 0, NULL) == -1) {
                close(kq);
                return(ret);
        }
        while(not_done) {
                ret = kevent(kq, NULL, 0, &event, 1, NULL);
                if (ret == 1) {
                        /* got the event */
                        not_done = 0;
                        ret = 1;
                        continue;
                }
        }
        return(ret);
}
```

So using a kqueue seems pretty straightforward, but let's now look a bit deeper into how sockets and kqueue interacts as well as conside-r ations we must make for a multi-threaded event loop.

Kqueue, Socket Calls, Multi-threading and Other Mysteries

One of the first things you will want to do with a socket program is connect to a server using the connect(2) call. This, in the normal case, is a blocking call, which can take some time before it times out and gives you an error if the peer is somewhere out on the internet and a long way from you. Now to avoid this one would normally set non-blocking I/O and then do a select or poll for it. For our kqueue, we can do the same thing. The trick to remember on the connect call, however, is that you are **not** selecting on an EV_READ, but instead you are selecting on an EV_WRITE.

Now that I know this little fact seems obvious, you connect so you can write to the server some request right? Of course, when I first started playing with kqueues it was not quite that obvious and it was not until I had read a second (or was it a third) time through the man page that I stumbled onto that little bit of wisdom.

Another little socket trick is the placement of the kqueue call when on the server side you setup for a non-blocking listen. You **must** place the kqueue(2) call **before** you do the listen. If you do not do this, your listen will never wake up on your kqueue—even when it is full of pending sockets waiting to be accepted. This again was a, gee, ah-hah moment for me on a third read of the manual.

Another mystery for me was in my reading of Unangst there seemed to be an implication that if I got a read event and failed to read all I was told to read (the event .data field has the number of bytes ready to read when a socket wakes up a kqueue for reading), I might not be told to wake up again, unless some more data arrived. Of course, I worked real hard to make sure I always read the entire amount so that I would not get "stranded" by some slow sending socket. I even tried setting EV_CLEAR in the hopes of not having to worry about this "race condition." Setting EV_CLEAR actually turned into a disaster, however. This meant that **many** threads would wake up on that same event due to scheduling and context switching delay. Basically what most socket applications need is the EV_DISPATCH. Read what you want (or can) read and then when you want do a re-enable. In my testing with my little app (mentioned earlier) I proved that:

*Adding EV_CLEAR will stream in events until you either get the data read or shut off the event.

*Using EV_DISPATCH and reading only part of the message will result in another kqueue wakeup once the event is re-enabled.

EV_ERROR and EV_EOF were another set of mysteries. When did these wonderful flags get applied? Well, as it turns out, the EV_EOF will be returned once a FIN comes in from the other side. To be more succinct its really every time a kqueue event transpires on a socket and the socket has the SBS_CANTRCVMORE flag set against the receive socket buffer. What this means is that as long as you have data to read, you will continue to see the EV_EOF flag with every subsequent kevent. The EV_ERROR however is a bit different. You usually get these when the socket goes into an error state (ECONNRESET or ETIMEDOUT). Basically if the EV_ERROR gets set, the socket is pretty well washed up.

Now one other oddity, what I began seeing was an appearance of a wakeup on an event I had disabled. How could this occur? Was it that queued up events that had not been read might still be having an event that I would read later. This could cause me major issues since that little pointer in event.udata was being used to carry a pointer that I may have done a free upon. After continuing to see this happen under certain load conditions, I had to dig to the bottom of it. With my little test program I again proved that when you remove an event **all** unread events are removed for that socket (whew). So the oddity I had been seeing had to be something else, or did it?

If you remember the assignment I was given, a tcp proxy, any of the proxys will have two TCP descriptors for one flow. This is where my headache came from. Since it was entirely possible (but rare) that both socket descriptors would wake up at the same time and one of them would have a "reset" on it, if that socket descriptor got processed ahead of the other one, due to locking order of the two threads, disaster could strike. In effect, Thread1 would be destroying the flow, while Thread2 was patiently waiting to get the lock that Thread1 was holding on that flow. Thus, when the lock was released before destruction, Thread2 would wakeup and start accessing freed memory. This one was solved in a very interesting way, but I will leave it to the reader to puzzle out a solution.

Conclusion

So what conclusion can we draw from all of this madness?

- First and foremost, if you are going to play with kqueues its best to get completely familiar with them before diving in (either that or find better reading on the internet than I did on how to use them).
- Multi-threading with kqueues, especially those that have more than one file descriptor referencing the same object can provide some interesting challenges.
- There are some subtleties with kqueues in their interactions with sockets that need to be accounted for when writing the software (order is very important as well as which type of event you are interested in).
- The proper use of a kqueue can provide you with a very efficient program that can handle tasks with thousands of file descriptors with a minimal set of overhead.
- Understanding and testing the functionality of kqueues (with the before mentioned program) can surely help you from going mad.

References

[1] Kqueue: A generic and scalable event notification facility – Jonathan Lemon.
http://people.freebsd.org/~jlemon/papers/kqueue.pdf

[2] Experiences with kqueue – Ted Unangst, August 2009.
http://www.tedunangst.com/kqueue.pdf

Randall Stewart currently works for Adara Networks Inc. as a Distinguished Engineer. His current duties include architecting, designing and prototyping Adara's next generation routing and switching platform. Previously Mr. Stewart was a Distinguished Engineer at Cisco Systems. In other lives he has also worked for Motorola, NYNEX S&T, Nortel and AT&T Communication.

Throughout his career he has focused on Operating System Development, fault tolerance, and call control signaling protocols. Mr. Stewart is also a FreeBSD committer having responsibility for the SCTP reference implementation within FreeBSD.

Data Distribution With DNSSEC

BY MICHAEL W. LUCAS

One of the hard problems on a public network is trusting identity. How do you know a server is what it claims to be? Within an organization you can have many different solutions, but on the public Internet you have the choice of trusting assorted external entities to verify identity (Certificate Authorities) or hand-crafting your own network of identification (OpenPGP's web of trust). DNS Security Extensions, or DNSSEC, is changing that.

What Is DNSSEC?

The Domain Name Service (DNS) maps hostnames to IP addresses and vice versa. It's why you can type http://www.FreeBSD.org into your web browser, rather than http://8.8.178.110. But DNS was designed back in the early 1980s, and was intended for a much smaller network with a much more limited range of users. At the time, people were just happy to have a protocol that let system administrators enter a hostname and get an IP back. They didn't have to worry about spoofed addresses, malware, and financially motivated intruders. Today, on an Internet riddled with criminals and malware, we need a way to verify that DNS information is accurate.

DNS Security Extensions add cryptographic verification to DNS information. Each stage of the DNS lookup has a digital signature, and local clients can validate those signatures. DNSSEC does not provide confidentiality, as DNS is public data. But DNSSEC makes changes obvious—it makes DNS tamper-evident to anyone who cares to look.

In some ways, DNSSEC combines the Web of Trust with Certificate Authorities. The root zone has a digital signature. Clients must trust that signature. (Yes, there are protocols to replace that signature as in case of compromise.) The root zone gives digital signatures to records for the zones beneath it, and those zones sign the records for zones beneath them. So when you use DNSSEC to look up the IP address for www.FreeBSD.org, the client can get the digital signature on the host www.freebsd.org, and then use the signature inside .org to validate the domain's signature, and finally use the signature on the root zone to validate .org. The client builds a chain of trust, using the trusted key on the root zone to trust the identity of the end site.

If you have to trust an external entity, how does DNSSEC differ from a certificate authority? Go look in your web browser. You'll see dozens of certificate authorities. Some of these organizations are more careful and reputable than others. Many of them have issued certificates to organizations not eligible for them – for example, more than one CA has issued certificates for companies like Google, Apple, and Microsoft to organizations or individuals that shouldn't have them.

Organizations, even careful CAs, fail. With DNSSEC, you don't rely on a third party to authenticate identity.

Also, certificate authorities expect payment for validating identity. That payment might not be much, but when you have dozens or hundreds of servers it can become rather pricey. Many organizations choose to not secure data in transit rather than spend money securing every little connection.

Today, DNS is used for many things besides host and IP address information. DNS offers configuration information to anything from servers to phones, lists valid mail senders for a domain, and more. Once this channel becomes tamper-evident, though, you can start to put public security information in DNS. We'll consider two types of data commonly distributed via DNSSEC: fingerprints for SSH public keys and SSL certificate.

SSH Host Key Fingerprints

SSH uses public key cryptography to verify that the server you're connecting to is actually the server you think it is, and to secure the data exchange between client and server. Correct use of SSH requires that when a user first connects to a server, he must examine the offered public key and compare it to an accurate copy of the public key. This is tedious and annoying, and most SSH users don't bother. Worse, most SSH users quickly decide to ignore public key warnings from SSH, reducing the protocol's security.

Comparing two cryptographic fingerprints is exactly the sort of thing a computer is good at, but until now there was no standard way to securely transmit these fingerprints online. DNSSEC provides that channel through SSH Fingerprint (SSHFP) records. Newer versions of FreeBSD include an OpenSSH client that automatically checks for SSHFP records.

Start by creating SSHFP records for your hosts and insert them into the zone's record for that host.

Run `ssh-keygen –r` to generate SSHFP records from the public key files in /etc/ssh. Use the hostname, as you want it to appear in the SSHFP record, as an argument. Here I create SSHFP records for my web server, www.michaelwlucas.com.

```
$ ssh-keygen -r www
www IN SSHFP 1 1 f44d08efc159…
www IN SSHFP 1 2 86c744ce05ba…
www IN SSHFP 2 1 9af675d68969…
www IN SSHFP 2 2 7914036e9053e14db552…
www IN SSHFP 3 1 0f7c928e3954c54f0b32…
www IN SSHFP 3 2 5c5192e78de10…
```

This displays the SSHFP records for all host keys on the local machine. These records are specific to the local machine—I could run the exact same command on www.FreeBSD.org and would get records with completely different fingerprints.

If a host has multiple names and you might use any of those names to connect to the host, then each host name needs SSHFP records. For example, my web server is also known as pestilence.michaelwlucas.com. I'll need two copies of these records in my zone, one for each hostname, creating something like this.

```
www IN A 192.0.2.33
www IN SSHFP 1 1 f44d08efc159…
www IN SSHFP 1 2 86c744ce05ba…
www IN SSHFP 2 1 9af675d68969…
www IN SSHFP 2 2 7914036e9053e14db552…
www IN SSHFP 3 1 0f7c928e3954c54f0b32…
www IN SSHFP 3 2 5c5192e78de10…
pestilence IN A 192.0.2.33
pestilence IN SSHFP 1 1 f44d08efc159…
pestilence IN SSHFP 1 2 86c744ce05ba…
```

The hashes are identical for each variant of the hostname.

To make an OpenSSH client check for SSHFP records, set VerifyHostKeyDNS to yes in ssh_config or ~/.ssh/config. ssh(1) will use the SSHFP records to validate host keys without prompting the user. Computers are good at comparison. You aren't. Let them do the work.

DANE and TLSA

DANE, or DNS-based Authentication of Named Entities, is a protocol for stuffing public key and or public key signatures into DNS. As standard DNS is forged easily, you can't safely do this without DNSSEC. With DNSSEC, however, you now have an alternative way to verify public keys. We'll use DNSSEC-secured DNS to verify web site SSL certificates (sometimes called DNSSEC-stapled SSL certificates).

In DNSSEC Mastery I predicted that someone would release a browser plug-in to support validation of DNSSEC-staples SSL certificates. This wasn't terribly prophetic, as several different groups had already started down that road. I'm pleased to report that the fine folks at http://dnssec-validator.cz have completed their TLSA verification plugin. I'm using it in Firefox, Chrome, and IE. One day browsers will support DANE automatically, but until then, we need a plug-in.

DNS provides SSL certificate fingerprints with a TLSA record. (TLSA isn't an acronym; it's just a TLS record, type A. Presumably we'll move on to TLSB at some point.) A TLSA record looks like this:

```
_port._protocol.hostname TLSA ( 3 0 1 hash...)
```

If you've worked with services like VOIP, this should look pretty familiar. For example, the TLSA record for port 443 on the host dnssec.michaelwlucas.com looks like this: long one

```
_443._tcp.dnssec TLSA ( 3 0 1
4CB0F4E1136D86A6813EA4164F19D294005EBFC02F10CC400F1776C45A97F16C)
```

Where do we get the hash? Run openssl(1) on your certificate file. Here I generate the SHA256 hash of my certificate file, dnssec.mwl.com.crt.

```
# openssl x509 -noout -fingerprint -sha256 < dnssec.mwl.com.crt
SHA256 Fingerprint=4C:B0:F4:E1:13:6D:86:A6:81:3E:A4:16:4F:19:D2:94:00:5E:BF:C0:2F:10
:CC:40:0F:17:76:C4:5A:97:F1:6C
```

Copy the fingerprint into the TLSA record. Remove the colons. That's it.

Interestingly, you can also use TLSA records to validate CA-signed certificates. Generate the hash the same way, but change the leading string to 1 0 1. I'm using a CA-signed certificate for https://www.michaelwlucas.com, but I also validate it via DNSSEC with a record like this.

```
_443._tcp.www TLSA ( 1 0 1
DBB17D0DE507BB4DE09180C6FE12BBEE20B96F2EF764D8A3E28EED45EBCCD6BA )
```

So: if you go to the trouble of setting this up, what does the client see?

Start by installing the DNSSEC/TLSA Validator (https://www.dnssec-validator.cz/) plugin in your browser. Hopefully, by the time this article reaches you it will be an official FreeBSD port. In the meantime, Peter Wemm has built the Firefox version of the plugin on FreeBSD, and he has a patch (http://people.freebsd.org/~peter/MF-dnssec-tlsa_validator-2.1.1-freebsd-x64.diff.txt) and a 64-bit binary. (http://people.freebsd.org/~peter/MF-dnssec-tlsa_validator-2.1.1-freebsd-x64.xpi) If you're looking for a way to contribute to FreeBSD, porting this would be very useful.

The plugin adds two new status icons. One turns green if the site's DNS uses DNSSEC, and has a small gray-with-a-touch-of-red logo if the site does not. Not having DNSSEC is not cause for alarm. The second icon turns green if the SSL certificate matches a TLSA record, gray if there is no TLSA record, and red if the certificate does not match the TLSA record.

Should you worry about that self-signed certificate? Check the TLSA record status. If the domain owner says "Yes, I created this cert," it's probably okay. If the self-signed cert fails TLSA validation, it's a self-signed certificate: probably okay on a mailing list archive, not okay for your bank.

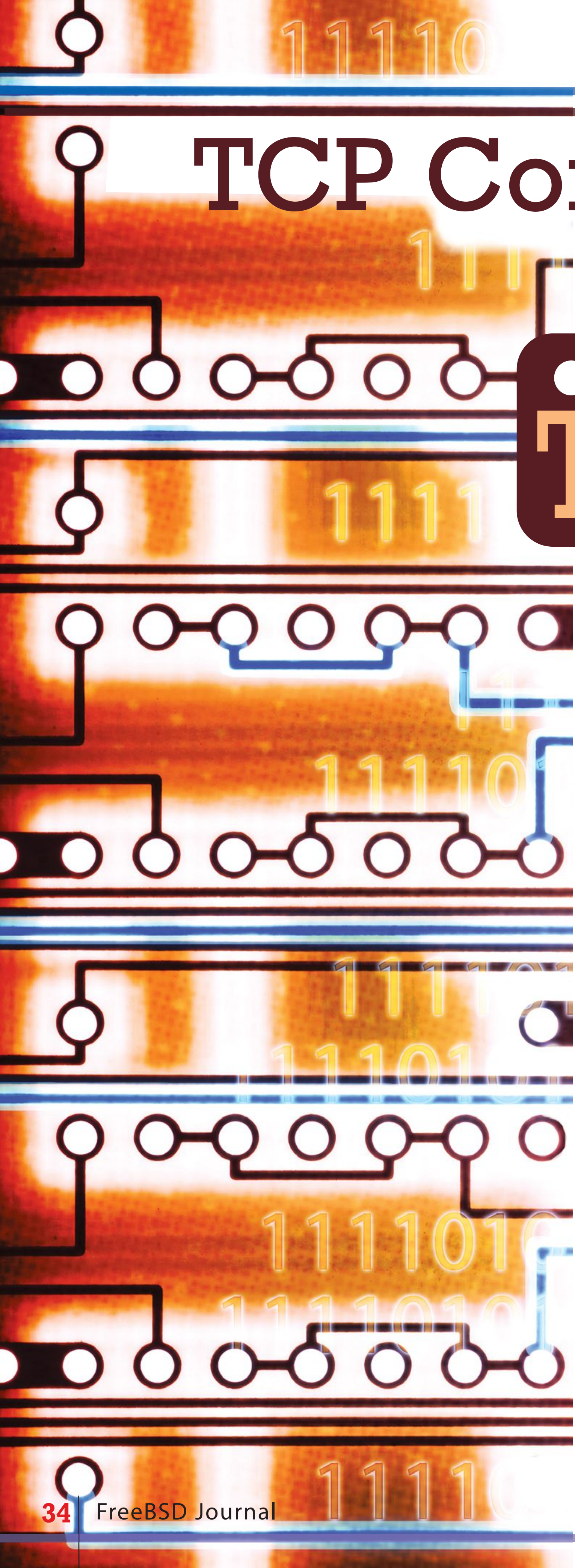
You can use a variety of hashes with TLSA, and you can set a variety of conditions as well. Should all certificates in your company be signed with RapidSSL certs? You can specify that in a TLSA record. Do you have a private CA? Give its fingerprint in a TLSA record. If you want to play with these things, check out RFC 6698 or my book "DNSSEC Mastery."

I have had some issues with the plugin on my laptop after suspending it. My home and office both perform DNSSEC validation. When I travel to the coffee shop and resume, the plug-in causes performance issues. Restarting the browser solves them. I expect this to improve quickly, however, and it might be a non-issue before you read this article.

DNSSEC gives you an alternate avenue of trust, outside of the traditional and expensive CA model. Spreading TLSA more widely means that you can protect more services with SSL without additional financial expenses.

Of course, you cannot deploy either of these services without working DNSSEC. DNSSEC isn't that hard these days—if I can do it, you can too.

Michael W. Lucas is the author of *Absolute FreeBSD, Absolute OpenBSD,* and *DNSSEC Mastery,* among others. He lives in Detroit, Michigan, with his wife and a whole mess of rats. Visit his website at https://www.michaelwlucas.com.



TCP Connection Rat

By Michael Bentkofsky and Julien Charbon

Today's commodity servers, with bandwidth of 10+ Gigabits per Network Interface Card (NIC) port and dozens of processor cores, have sufficient network and processing capacity to host the most demanding network services on a small server footprint.

With the popularity of web-based services, significant attention has gone into scaling these types of services to address issues such as the C10K problem (http://en.wikipedia.org/wiki/C10k_problem) of the previous decade that posed the challenge of handling 10,000 simultaneous connections on a single server.

Modern server software that handles tens of thousands of simultaneous connections is implemented using non-blocking I/O and event notification such as kqueue(). Today's new challenge, however, is aimed at servicing up to a million connections concurrently on a single server. Current generation NIC hardware can support this, but in order to keep scaling connection counts higher, one of the remaining challenges is to scale the TCP connection rate that can be serviced by a single server (http://blog.whatsapp.com/index.php/2012/01/1-million-is-so-2011/). This article considers several types of TCP-based services where the primary scaling problem is handling the highest rate of TCP connection establishment on modern server hardware. This is a different challenge from serving content to millions of established TCP connections. Examples of such services include:

- DNS services over TCP (http://tools.ietf.org/html/rfc5966),
- HTTP services with a single request and response, such as an Online Certificate Status Protocol (OCSP) service,
- Whois services for domain name registries.

e Scaling in FreeBSD

Of the examples above, DNS over TCP is the type of traffic we should expect to increase on the Internet as the adoption of DNSSEC causes larger DNS responses (https://www.dns-oarc.net/node/199) and as service providers use TCP to reduce attack vectors that are associated with connectionless protocols such as User Datagram Protocol (UDP). To delve into this scaling challenge, Figure 1 shows the packets associated with establishing a TCP connection, exchanging the request and response data, and tearing-down the TCP connection.

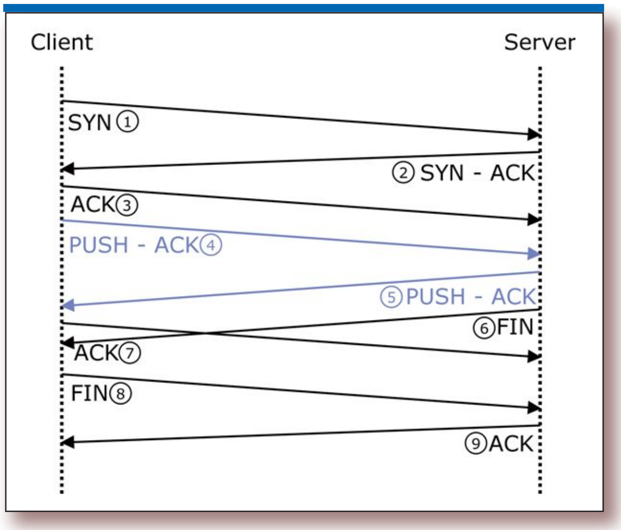


Figure 1. Typical packets, small TCP requests

In this use case, a client is initiating a connection SYN (1), the server responds with the SYN/ACK (2), and the client establishes the connection with an ACK (3). The request data packet from the client to the server is sent in (4) with a response data packet from the server in (5). This represents the exchange of a request to the server and a response sent back to the client. The server initiates the connection teardown (6) while the response is acknowledged (7). The client tears down its side of the connection (8) and the server finally acknowledges the client's FIN (9). Either end can initiate the connection teardown, so packets (6), (7), and (8) could be slightly differently sequenced. In this packet flow we can make several observations:

- There are a total of nine packets exchanged with five sent from the client to the server and four sent from the server to the client.
- There is a fair bit of overhead in establishment and teardown of connections with only two packets of data exchanged between the machines in (4) and (5). Although there can be more data exchanged on the same connection, this workflow considers small requests and responses.
- This kind of communication will typically have primarily small packets exchanged, depending on the size of the request and response. For this article, we'll consider generally small requests and responses, to analyze the worst-case scenarios to scale these types of exchanges.

This packet exchange is considered the canonical form of the small request and response problem for TCP. While the packet flow can be slightly different, such as the client and server both simultaneously initiating connection teardown, or the FIN being sent along with the data packet from either end, those differences do not significantly alter the nature of the scaling.

For this packet flow, we'll consider what it would take to scale to the NIC bandwidth capacity of currently available cards. For each packet, there will be at least 78 bytes:

- 12-byte inter-frame gap
- 8-byte preamble + the start of frame delimiter
- 18-byte Ethernet (source and destination MAC, type, checksum)
- 20-byte IP header (IPv4 without options)
- 20-byte TCP header at a minimum

If we assume that bandwidth will be limited primarily by egress (response) packets, for the canonical packet flow, we can calculate per request egress bandwidth consumption:

Packet Type	Octets	Notes
(2) SYN - ACK	98	78 + MSS, SACK, wscale, timestamps
(5) PSH - ACK	90+	78 + timestamps + response payload
(6) FIN	90	78 + timestamps
(9) ACK	90	78 + timestamps
Total Egress	368 + response payload	

```
$ top -SHIPz

CPU 0:   0.0% user,  0.0% nice,  0.4% system, 96.1% interrupt,  3.5% idle [irq core]
…
CPU 4:   2.4% user,  0.0% nice, 20.0% system,  0.0% interrupt, 77.6% idle [accept thread]
CPU 5:  11.8% user,  0.0% nice, 63.5% system,  0.0% interrupt, 24.7% idle [worker thread}
…

  PID  USERNAME   PRI  NICE   SIZE    RES  STATE   C   TIME     WCPU   COMMAND
   12  root       -92     -     0K   560K  CPU0    0   8:39   75.78%   [intr{irq263: ix0}]
 2636  ...         85    -5  4927M  4886M  CPU5    5   9:16   58.06%   tcp_server
 2636  ...        -21   r31  4927M  4886M  CPU4    4   3:00   20.26%   tcp_server
```

Figure 2. CPU Utilization, Single Queue

Additional payload, VLAN-tagging (802.1Q), slightly different packet counts, and additional TCP options could increase the cumulative request size, so if we assume aggregate inbound or outbound packet sizes between 500 and 1,000 bytes, a single gigabit NIC port should strive to serve between about 125,000 and 250,000 of these requests cumulative per second, representing between 625,000 and 1.25 million inbound packets per second and between 500,000 and 1 million outbound packets per second.

We measure how a multi-core server handles this workload without any overhead introduced in the application. The application is a simple TCP server running under FreeBSD 10.0, with a user space application listening on a socket in one thread, assigning file descriptors for new connections to multiple other threads, termed “worker threads.” All threads are cpu_set to separate cores and there are not more user threads than total cores. The worker threads service multiple connections with kqueue(), read the request data with a single receive, generate the response with a single send, and close the connection.

For the client side, multiple requests are generated from a private network using a wide range of client source IP addresses and ports. While the client capacity is out-of-scope for this article, the client software has been benchmarked to create hundreds of thousands of connections per second.

At approximately 56,000 connections per second, or between 225 – 450 Mbits/sec of bandwidth, we see the CPU utilization shown in Figure 2. The bottleneck on core 0 is due to interrupt

```
CPU 0:  0.0%   user,  0.0%   nice,   30.5%   system,  64.5%   interrupt,   5.0%  idle  [irq core]
CPU 1:  0.0%   user,  0.0%   nice,   29.9%   system,  64.3%   interrupt,   5.7%  idle  [irq core]
CPU 2:  0.0%   user,  0.0%   nice,   30.2%   system,  66.1%   interrupt,   3.6%  idle  [irq core]
CPU 3:  0.0%   user,  0.0%   nice,   30.2%   system,  66.4%   interrupt,   3.4%  idle  [irq core]
CPU 4:  5.7%   user,  0.0%   nice,   84.4%   system,   0.0%   interrupt,   9.9%  idle  [accept() thread]
...
CPU 6:  8.3% user,    0.0%   nice,   59.1%   system,   0.0%   interrupt,  32.6%  idle  [worker thread]
CPU 7:  8.9% user,    0.0%   nice,   58.3%   system,   0.0%   interrupt,  32.8%  idle  [worker thread]
...

    PID  USERNAME   PRI  NICE   SIZE     RES    STATE   C     TIME     WCPU   COMMAND
   3110  ...        -21   r31  4955M   4911M    CPU4    4     5:12   70.26%   tcp_server
     12  root       -92     -     0K    688K    CPU1    1     7:20   66.46%   intr{irq264: ix0:que }
     12  root       -92     -     0K    688K    CPU2    2     7:21   66.26%   intr{irq265: ix0:que }
     12  root       -92     -     0K    688K    CPU3    3     7:21   66.16%   intr{irq266: ix0:que }
     12  root       -92     -     0K    688K    CPU0    0     7:20   65.87%   intr{irq263: ix0:que }
   3110  ...         52    -5  4955M   4911M    RUN     7     4:50   65.38%   tcp_server
   3110  ...         86    -5  4955M   4911M    CPU6    6     4:47   60.50%   tcp_server
      0  root       -92     0     0K    304K    RUN     1     0:50    8.06%   kernel{ix0 que}
      0  root       -92     0     0K    304K    RUN     0     0:49    8.06%   kernel{ix0 que}
      0  root       -92     0     0K    304K    RUN     3     0:48    8.06%   kernel{ix0 que}
      0  root       -92     0     0K    304K    RUN     2     0:49    7.96%   kernel{ix0 que}
```

Figure 3. CPU Utilization, Multiple Queues

```
# pmcstat -c 0 -S unhalted-cycles -O sample.pmc
# pmcstat -R sample.pmc -G call.graph

... [397427 samples]

64.82%       [257597]       __rw_wlock_hard @ /boot/kernel/kernel
99.01%       [255052]         tcp_input
100.0%       [255052]           ip_input
100.0%       [255052]             netisr_dispatch_src
00.91%       [2346]         in_pcblookup_hash
100.0%       [2346]           tcp_input
100.0%       [2346]             ip_input
00.07%       [180]          tcp_usr_attach
100.0%       [180]            sonewconn
100.0%       [180]              syncache_expand
00.01%       [19]           syncache_expand
100.0%       [19]             tcp_input
100.0%       [19]               ip_input

02.48%       [9851]         __rw_rlock @ /boot/kernel/kernel
26.34%       [2595]          vlan_input
...
```

Figure 4. PMC Profile, Interrupt CPU

processing for packets to and from the NIC on CPU 0. Fortunately this bottleneck can be overcome by using the NIC Receive-Side Scaling (RSS) feature which distributes the traffic across multiple queues and dispatches interrupt handling to multiple CPUs via Message Signaled Interrupts (MSI or MSI-X).

With RSS, a NIC is configured to have multiple receive hardware queues, and MSI allows interrupts from the NIC to be directed to particular CPU cores that are lightly loaded. The NIC directs packets to receive queues based upon a hash function of packets, often a hash of the source and destination IPs and ports. In FreeBSD, this configuration is set partially in the device driver initialization, and then interrupt processing can be pinned to particular cores using cpuset. With a properly configured driver and smart selection of CPUs to handle interrupts, we should expect to see the processing workload shift more to the lightly loaded cores.

If all packet processing could be done in parallel, we would expect to see the capacity scale linearly with the number of receive queues and CPUs dedicated to receive processing, until the next bottleneck was reached. The results from such a test however are inconsistent with that, as shown in Figure 3, with 62,000 connections per second, or 250 to 500 Mbits/sec of bandwidth. In this configuration there are four NIC queues with affinity to four CPUs. An additional thread is added in the user space application to handle requests since Figure 2 suggested that CPU 5 was highly loaded. We note that interrupt processing has consumed all of the four dedicated CPUs and there are indications that both the threads accepting new connections and the two threads handling requests consume significant system time.

To analyze why distributing the TCP input processing workload on multiple MSI receiving queues did not scale as expected, we used Performance Monitoring Counter (PMC) profiling on one of the CPUs handling interrupt processing, as shown in Figure 4.

According to this profile, more than 50 percent of the CPU time of this core was spent in __rw_wlock_hard() and this kernel function was predominantly called from tcp_input().

__rw_wlock_hard() is part of the reader/writer kernel lock implementation [rwlock(9)], and more precisely this function aims for an exclusive access on the lock. Details about contended locks have been provided by running kernel lock profiling [LOCK_LOCK_PROFILING(9)] and sorting results by the total accumulated wait time for each lock (wait_total) in microseconds, shown in Figure 5.

The main contention point for this workload is on the rw:tcp lock. If we look at the top call points for this lock, we see in rank order:

1. sys/netinet/tcp_input.c:1013: tcp_input()
2. sys/netinet/tcp_input.c:778: tcp_input()
3. sys/netinet/tcp_usrreq.c:635: tcp_usr_accept()
4. sys/netinet/tcp_usrreq.c:984: tcp_usr_close()
5. sys/netinet/tcp_usrreq.c:728: tcp_usr_shutdown()

The first two locks in tcp_input() are called by the kernel on each TCP packet whereas the latter three calls correspond to socket system calls from the user-space TCP server. The per-packet lock is acquired in tcp_input() under these conditions:

```
# sysctl debug.lock.prof.stats | head -2; sysctl debug.lock.prof.stats | sort -n -k 4 -r | head -6

max  wait_max     total   wait_total     count    avg wait_avg cnt_hold cnt_lock name

210       660   2889429      7854725    568650     5      13      0   562770  …/tcp_input.c:1013 (rw:tcp)
 79       294   3346826      7309124    642543     5      11      0   541026  …/tcp_input.c:778 (rw:tcp)
  9       281    109754      4907003    321270     0      15      0   284203  …/tcp_usrreq.c:635 (rw:tcp)
  6       204    174398      1484774    321267     0       4      0   284754  …/tcp_usrreq.c:984 (rw:tcp)
 17       207   1252721      1195551    321268     3       3      0   241000  …/tcp_usrreq.c:728 (rw:tcp)
210       197   3828100       315757   1606362     2       0      0    90016  …/in_pcb.c:1802 (rw:tcpinp)
```

Figure 5. Lock Profiling the TCP Workload

- Any of the SYN, FIN, or RST TCP flags are set
- The TCP connection state is any state other than ESTABLISHED

Looking at the packets exchanged and corresponding TCP connection states (shown in the table below), we see that four of the five packets received cause the exclusive write lock rw:tcp to be acquired. As this lock is global to all TCP sockets, this contention appears to be a large factor in limited scaling through RSS.

#	Packet In	Packet Out	State, Before	State, After	rw:tcp locked in tcp_input()?
1	**SYN** (1)	SYN + ACK (2)	None	SYN-RECEIVED	**Yes, WLOCK**
2	ACK (3)		**SYN-RECEIVED**	ESTABLISHED	**Yes, WLOCK**
3	PSH (4)	PSH (5):	ESTABLISHED	ESTABLISHED	No
4		FIN (6)	ESTABLISHED	FIN-WAIT-1	
5	ACK (7)		**FIN-WAIT-1**	FIN-WAIT-2	**Yes, WLOCK**
6	**FIN** (8)	ACK (9)	**FIN-WAIT-2**	TIME-WAIT	**Yes, WLOCK**

In Figure 6, we see that the contention points on rw:tcp include tasks originated from interrupts driven from the NIC and packet processing, from user space system calls, and from other periodic timer tasks such as the TCP TIME-WAIT timer.

The rw:tcp lock protects the global data structures defined for the TCP state including:

The hash table (struct inpcbinfo.ipi_hashbase) to search among the structures (struct inpcb)

The global list to scan the structures (struct inpcbinfo.ipi_listhead)

Layer 4 specific additional structures including the list of sockets in the TIME-WAIT state for TCP.

The lock is defined as a readers-writer lock so multiple read tasks may be simultaneously searching, but only one task may be updating while also blocking all readers. When an inpcb structure is added or removed, the writers lock is held. In the context of TCP this occurs when a connection is established with ACK(3) that completes the connection and when the connection is entirely torn down and TIME-WAIT has completed.

In addition to the global rw:tcp lock, each inpcb has a lock named rw:tcpinp. This lock is held when per-connection information is updated. With long-running connections, each rw:tcpinp lock may be held and released several times, although this only causes per-connection contention, not across all connections. While rarely causing contention for short-lived connections, multiple locks necessitate a well-established lock order to avoid deadlock. This well-established order is rw:tcp (the global lock) must be locked before a rw:tcpinp (the per-connection lock) is locked when both locks are to be held. As previously noted, the TCP state transitions on four of the five received packets lead to both locks being held.

There are cases where the rw:tcp lock is held other than normal packet processing considered here and the user-space system calls. Other examples include inbound packets with the TCP RST flag set, conditions where resources can't be allocated or system configured limits are reached, and even unusual cases such as when the TCP congestion control algorithm is reconfigured. Careful attention must also be given to these other processing paths to avoid possible deadlock or global structure corruption whenever considering changes to the locking strategy to scale connection rate. While complex, there appear to be several opportunities to reduce contention on the global rw:tcp lock. Analyzing the paths where the lock is acquired shows sev-

TM

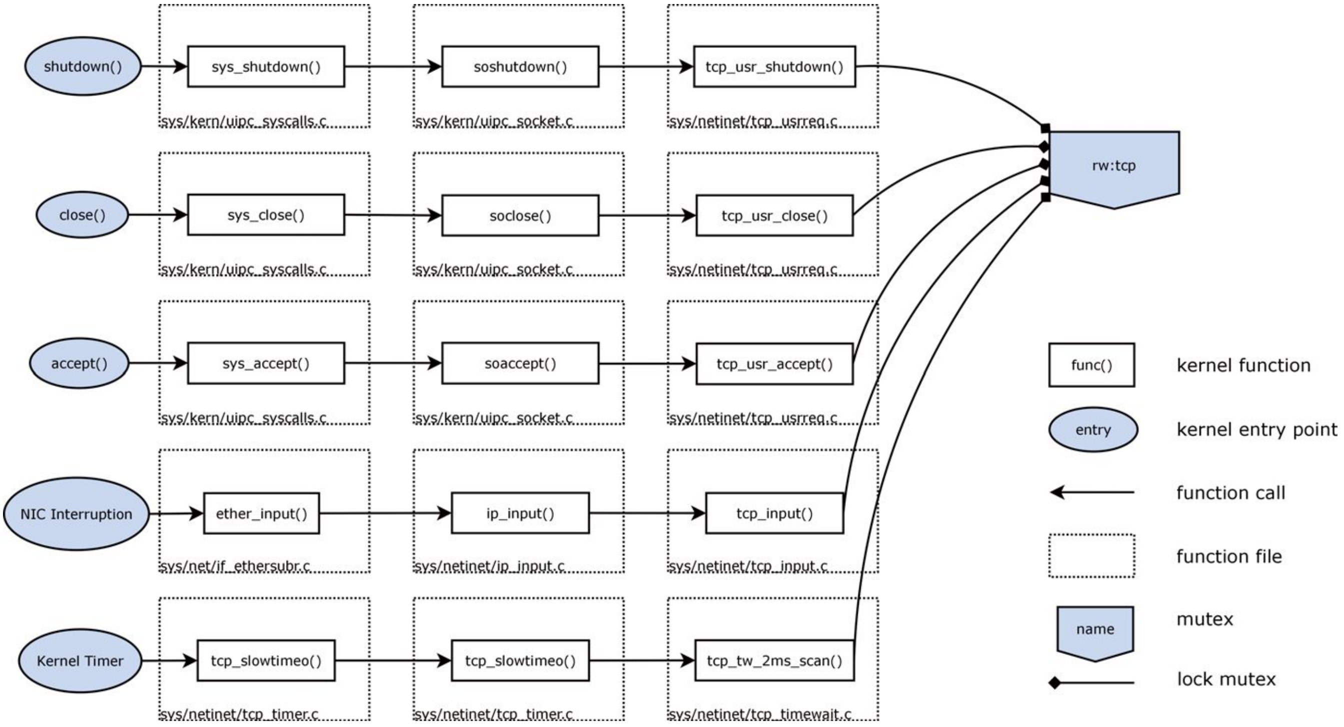


Figure 6. Lock Contention Points for rw:tcp

eral promising ways to:
1. Avoid the rw:tcp lock altogether, particularly driven from user-space system calls,
2. Add new finer-grained locks where the rw:tcp lock is currently used,
3. Switch to rw:tcp read locks to permit concurrency in critical code paths.

As an example of the first contention mitigation approaches, an implementation that avoids locking rw:tcp lock during the accept() system call has been adopted for future versions of FreeBSD (http://www.freebsd.org/cgi/query-pr.cgi?pr=183659). There appear to be other similar opportunities.

As an example of the second mitigation approach, an alternative implementation is also being adopted for expiring connections in the TIME-WAIT state (http://svnweb.freebsd.org/base?view=revision&revision=264321). Instead of locking the rw:tcp lock to manage the global TIME-WAIT list, a new lock rw:tcptw is created. While the rw:tcp lock is still used to finally destroy the inpcb structures, it is only held briefly and not while iterating the expiration list.

Early performance results using the two cited patches show that the techniques help scale connection rate. With accept() system call avoidance and a separate fine-grained lock for TIME-WAIT, we see the connection-rate increase from 62,000 connections per second to 69,000 connections per second. While modest, there appear to be further applications of these two techniques.

This work will continue to explore the possibility of allowing more parallelism in packet processing. Although in early development, the third technique looks promising. Early performance results suggest this technique could exceed 120,000 connections per second utilizing 480 – 960 Mbits / sec of bandwidth.

While high connection rates may be atypical for many networked services including streaming media or services with long-standing connections, such applications are present. The challenge of handling a high rate of connections that could utilize a single Gigabit NIC is within reach, although more work needs to be done to achieve that goal. ●

Julien Charbon is a Software Development Engineer at Verisign, Inc. Julien has worked on the company's high-scale network service ATLAS platform and related high-scale network services. Julien has worked with FreeBSD to perform tasks including porting software, developing fixes and patches, and network performance studies.

Michael Bentkofsky is a Principal Software Engineer at Verisign, Inc. and leads the development of their ATLAS platform. He has been part of teams implementing high-scale, always-available TCP services including DNS and Whois services for the .COM and .NET top-level domains, and the Online Certificate Status Protocol (OCSP) for certificate validation.

Testers, Systems Administrators, Authors, Advocates, *and of course* **Programmers** *to join any of our diverse teams.*

COME JOIN THE PROJECT THAT MAKES THE INTERNET GO!

★ DOWNLOAD OUR SOFTWARE ★

http://www.freebsd.org/where.html

★ JOIN OUR MAILING LISTS ★

http://www.freebsd.org/community/mailinglists.html?

★ ATTEND A CONFERENCE ★

• *http://www.**bsdcan.org**/2014/* • *http://2014.**eurobsdcon.org**/*

The FreeBSD Project

The FreeBSD FOUNDATION

PORTSreport

BY THOMAS ABTHORPE

Ports Report is a summary of recent activity in the Ports infrastructure, news in the world of Ports, tips and tricks.

NEW PORTS COMMITTERS

As a dedicated contributor to the Ports Tree, other committers sit up, take notice and opt to propose you for a commit bit so they won't have to do your work for you anymore [sic]. Our most recent addition to the ranks of Ports Committers is Jost Meixner. One of Jost's many interests is maintaining Linux ports for FreeBSD.

NEWEST MEMBERS TO portmgr-lurkers

With the ongoing success of the portmgr-lurker project, we have started the second intake of lurkers to learn, observe, and contribute to the Ports Management Team. In March, Alexy (danfe@) Dokuchaev and Frederic (culot@) Culot commenced their duties. Frederic is also doing double duty, shadowing the portmgr-secretary and helping with the duties of that position.

THE SECOND BRANCH OF THE PORTS TREE

Because the first—2014Q1—branch was experimental, you might not have heard of it as yet. January 2014 saw the release of the first quarterly branch, intended to provide a stable and high-quality ports tree. Those stable branches are snapshots of the head ports tree taken every three months and currently supported for three months,

DOING YOUR PART TO IMPROVE THE PORTS TREE

Companion tools to the build systems of ports-mgmt/tinderbox and ports-mgmt/poudriere mentioned above are ports-mgmt/porttools. With these tools, you can use it to create a new port, file a PR for an update via send-pr(1), or even use it as a poor man's build system by issuing the comand port test. You can read more at http://www.freebsd.org/doc/en/books/porters-handbook/testing-porttools.html.

When you install the tools, you get another great porter's tool called portlint(1). Just as lint(1) aids you in getting the fluff out of your C program, portlint uses heuristics to assist you in finding errant whitespace, incorrectly placed directives, along with a host of other tips and suggestions to improve your port. For instance issuing

```
portlint -C /usr/ports/devel/fakeport # this is an example only
```

might yield something like

```
WARN: Makefile: [14]: possible direct use of command “env” found. use ${SETENV} instead.
WARN: Makefile: only one MASTER_SITE configured. Consider adding additional mirrors.
WARN: Makefile: “RUN_DEPENDS” has to appear earlier.
WARN: Consider to set DEVELOPER=yes in /etc/make.conf
0 fatal errors and 5 warnings found.
```

So I take the advice and add DEVELOPER=yes to /etc/make.conf. This, in itself, does not aid portlint, but gives some developer-directed verbose output that is useful when you build a port.

The other warnings need to be scrutinized by you, the porter, to see if additional action needs to be taken. Just remember we control portlint, it does not control us. Its warnings are intended as a guideline only.

The port tree is a collection of value-added software for FreeBSD. It is a collaborative effort by contributors around the world, and each one has done a little something to make it just a little better. A big "Thank You" to all who have helped out.

- http://fb.me/portmgr — "Like" us
- http://twitter.com/freebsd_portmgr — Follow us
- http://blogs.freebsdish.org/portmgr/ — Our blog
- https://plus.google.com/u/0/communities/108335846196454338383 — G+1 us

during which they receive security fixes as well as build and runtime fixes. Packages are built on a regular basis on that branch (weekly) and published as usual via pkg.FreeBSD.org (/quarterly, instead of the usual, /latest). On April 1 (no joke), the 2014Q2 branch was created and the first builds from it began shortly thereafter.

TIPS
FOR PROSPECTIVE PORTERS

Use a (semi)-automated build system to test your ports. In a previous issue, we recommend ed subscribing to http://redports.org to test your ports. It is available for all to use for the common good. Some people may have the necessary hardware resources to build their own build system. If you are one of them, then there are some tools in the trees you can try. The original build system is the venerable Tinderbox, found in the tree as ports-mgmt/tinderbox, or its bleeding edge counterpart, ports-mgmt/tinderbox-devel. You can read more about it at their website http://tinderbox.marcuscom.com/. In recent years, a newer build system came into being called Poudriere, which loosely translated is simply French for tinderbox. It can be found in the ports tree as ports-mgmt/poudriere along with its bleeding-edge version ports-mgmt/poudirere-devel. You can read more about it at their website, http://fossil.etoilebsd.net/poudriere. The poudriere build system is now the foundation for the Ports Management Team to do -exp runs and package building.

Chose a build system that works for you, use it to your advantage to test build ports, verify that they install and uninstall cleanly, and even set up your own private packaging system.

svn UPDATE

by Glen Barber

It's that time of the year again—the FreeBSD Release Engineering Team has started the RELEASE CYCLE cycle for 9.3 RELEASE. This edition of svn update covers what can be expected in 9.3-RELEASE as well as highlights for other active branches in the FreeBSD source tree.

ZFS Boot Support for Bhyve: head@r262331

(http://svnweb.freebsd.org/base?view=revision&revision=r262331)

The bhyve(8) hypervisor now has support for booting the virtual machine from a ZFS dataset allowing pure "root on ZFS" virtual machines. Prior to this change, it would have only been possible to boot from a UFS file system.

urndis(4) Support: stable/9@r262362

(http://svnweb.freebsd.org/base?view=revision&revision=r262362)

The urndis(4) driver has been added to stable/9. This driver was originally available in head/ with revisions r261541, r261543, and r261544, and provides Ethernet access over Remote NDIS, allowing mobile devices such as phones and tablets to provide network access via USB tethering.

The urndis(4) driver is also available in stable/10 as of revision r262363. The urndis(4) driver should support any USB RNDIS provider, such as those found on Android devices. The urndis(4) driver was ported from OpenBSD.

ext4 File System Support: stable/9@r262564

(http://svnweb.freebsd.org/base?view=revision&revision=r262564)

The ext2fs(5) code has been updated to enable ext4 file system support in read-only mode. Support for mounting read-only ext4 file systems was added in revision r262346 in head/.

Support for read-only ext4 file system is also available in stable/10 as of revision r262563.

BIND Release Update: stable/9@r262706

(http://svnweb.freebsd.org/base?view=revision&revision=r262706)

The BIND DNS server in the stable/9 base system has been updated to version 9.9.5. The release notes for this software update can be found here: https://lists.isc.org/pipermail/bind-announce/2014-January/000896.html.

pkg(8)Boostrap Signature Verification: stable/9@r263038

(http://svnweb.freebsd.org/base?view=revision&revision=r263038)

Signature verification code for pkg(8) bootstrapping originally available in 10.0-RELEASE has been merged to stable/9. With this change, when the administrator issues the 'pkg bootstrap' command, the downloaded pkg(8) binary distribution will be verified against available keys directly included in the base system, providing a massive security benefit for administrators that do not wish to build the 'ports-mgmt/pkg' port to install the pkg(8) package management utility.

Radeon KMS Driver Support: stable/9@r263170

(http://svnweb.freebsd.org/base?view=revision&revision=r263170)

The Radeon KMS driver has been merged from head/. This supports Kernel Mode Setting (KMS) similar to the Intel KMS driver found on newer motherboards. This driver takes advantage of features available in newer Xorg drivers, such as xf86-video-ati.

VT Driver Merged: stable/9@r263817

(http://svnweb.freebsd.org/base?view=revision&revision=r263817)

The new system console driver, VT (also known as "NewCons"), has been merged from head/ to stable/9. The VT driver serves as a replacement for the sc(4) console driver, providing a number of enhancements ranging from UTF-8 font support, to enabling users running X11 with Kernel Mode

Setting ("KMS" for short) to switch back to console from X11.

The VT driver is also available in stable/10 as of revision r262861.

The VT driver was developed by Aleksandr Rybalko under sponsorship of The FreeBSD Foundation.

Support for FreeBSD/arm Releases: stable/10@r264106

(http://svnweb.freebsd.org/base?view=revision&revision=r264106)

Initial support for building FreeBSD/arm images as part of the release process has been merged to stable/10. This change will allow the FreeBSD Release Engineering Team to publish ARM images that may be written to an SD card with the dd(1) utility. At this time, support is added for BeagleBone Black, Raspberry Pi, PandaBoard, WandBoard Quad, and ZedBoard.

Support for FreeBSD/arm image builds was added to head/ in revision r262810, and uses Crochet, written by Tim Kientzle, as the backend build system.

Weekly images are published for the head/ and stable/10 branches and are available on the FreeBSD FTP mirror at:

ftp://ftp.FreeBSD.org/pub/FreeBSD/snapshots/ISO-IMAGES/arm/armv6/

This work was done by Glen Barber under sponsorship of The FreeBSD Foundation.

Compressed Release ISO Distribution: stable/9@r264246

(http://svnweb.freebsd.org/base?view=revision&revision=r264246)

As part of the release build process, installation images compressed with

xz(1) will now be available, reducing overall ISO download time for those with access to an xz(1) decompression utility. For the immediate timeframe, uncompressed images will remain available, so those that do not have access to an xz(1) decompression utility will not need to worry.

Compressed image support was initially added to head/ as of revision r264027, and merged to stable/10 as of revision r264245.

As a hobbyist, Glen Barber became heavily involved with the FreeBSD project around 2007. Since then, he has been involved with various functions, and his latest roles have allowed him to focus on systems administration and release engineering in the Project. Glen lives in Pennsylvania, USA.

this month In FreeBSD

BY DRU LAVIGNE

Many in the FreeBSD (and other BSD) communities mark the month of May on their calendars as the time to embark on the annual migration to Ottawa, Canada, for BSDCan. This conference, now in its 11th year, has grown over the years to include two days of tutorials, two days of three presentation tracks, Developer Summits, a Vendor Summit, Doc Sprints, Foundation meetings, and an opportunity to become BSD certified.

I recently asked Dan Langille, the organizer of BSDCan, for his thoughts on why and how BSDCan came to be and what he has learned about the BSD community along the way.

Dan writes:

“My first introduction to BSD conferences was FreeBSDCon at Berkeley in Oct 1999. I had been using FreeBSD for nearly 18 months by that time. Reading back on my notes (http://www.freebsddiary.org/freebsdcon99.php), I see I was impressed by the social nature of that conference and how it contrasted with others. The next year, I attended the same conference, but now known as BSDCon, at Monterey in 2000. I had another great time, lots of fun, and learning new stuff.

At the time, I was living in Ottawa and active with OCLUG, the local Linux group. I had helped them organize OSW (Open Source Weekend) which went rather well. If I recall correctly, BSDCon stopped running and nothing filled the void. I knew of Ottawa Linux Symposium, run by Andrew Hutton. I had lunch with Andrew a few times and we talked about running conferences. His conference was vastly different to BSDCan (e.g. the budgets were astounding). For comparison, one of the parties held back then cost more than entire budget of BSDCan 2014. It was this that gave me the confidence to talk to University of Ottawa about hosting a conference. I calculated that should the worst happen and nobody turned up, I'd be about $2,000 down. I registered bsdcan.org in August 2003 and everything went from there.

The announcement went out in Jan 2004 (http://lists.freebsd.org/pipermail/freebsd-announce/2004-January/000934.html) and a few months later, the BSDCan tradition started.

The most surprising thing was how well received the conference was and how relieved I felt once the conference was over. The applause at the start of the closing session was overwhelming and brought tears to my eyes. People said the conference went very well and they commented on how relaxed I was during the conference. They asked me what my secret was. I didn't have one. I just organized the conference to provide the type of experience I would want.

Things changed over the years. In the beginning, we went to the local pubs for lunch, but we soon outgrew what they could provide. Now lunch is included and people stay at the venue. That gives yet more face-time, and it's the social interaction that I saw at FreeBSDCon which lives on at BSDCan.

For the first conference, I wrote a registration package and a rough scheduling system. The registration software is still in use, but in 2007, we moved to Pentabarf for processing talk proposals and for generating the schedule. That was the same year I started running PGCon, which occurs the week after BSDCan.

In 2006, the FreeBSD project held a Developer Summit at BSDCan, and they have continued this each year since. It has grown to over 120 attendees and their participation has helped grow BSDCan.

Some comparisons between BSDCan and PGCon: although the two conferences are roughly the same size, PGCon receives roughly three times the submissions for talk proposals.

This year, I received a great deal of help from Jennifer Russell. She has been the main person collecting and arranging travel. It's been so successful that I want to continue involving more people in BSDCan organization, primarily for two reasons:

1- to reduce my work load

2- ensure continuity of BSDCan (I won't be doing this forever)

The main thing I always kept in mind when organizing a conference: stay core. Don't get involved in anything which is not a core conference activity. Stay focused. For example, if someone wants video, let them do it. Don't get involved.

The most surprising aspect of conference organization is the necessity of an optimistic attitude.

Treat your sponsors well. They are the life blood of the conference. Without their contributions,

BSDCan would not be as successful as it has been.
If you have never been to a BSD conference, pick one, and go. You are passionate about the tools you have chosen. Go to a conference and meet others with similar passion. The relationships you form will be extremely beneficial to both you and the projects you have chosen. There is no substitute for face to face meetings.”

On Dan’s last point, I couldn’t agree more. The relationships you form when attending a conference like BSDCan are a benefit that is hard to fully comprehend until you experience it. Many users are hesitant to attend a conference: sometimes they think the content will be over their heads, or that developers will look down on them for being just a user or that they won’t know what to say if they run into someone famous in the community. But do you know what new attendees quickly discover? That it is really cool to put faces to people that until now you have only dealt with as an IRC handle or an email address, and that the personality you may have envisioned is very different once you get to share a meal or chat face-to-face. And you learn that so-called famous people are really pretty laid-back and just another friendly someone you can chat with. That it is really cool to mention off-hand a problem that you have encountered only to find that the fix to that problem was committed by another attendee a few hours later. That it is amazingly refreshing to spend a few days with a whole bunch of people to whom you don’t have to explain what FreeBSD is and to realize that, while you may be the only FreeBSD person you know back home (not counting the ones to whom you have explained FreeBSD), you’re not the only FreeBSD person out there. And, possibly most surprisingly, comes the realization that while you may feel that your use of and contribution to FreeBSD is minimal, you really are a part of a larger community and that others are genuinely interested in how and why you are using FreeBSD. These are the reasons most attendees look forward to attending year after year.

MORE ON EASTER EGGS

- Dru Lavigne’s March/April column mentions Easter Eggs. It contains the comment "more is less than less and less is more than more." Whilst I agree with the latter, more(1) predates less(1) by at least 4 years.
- Rummaging around, I note the earliest Unix more(1) I have is from 3BSD and dated November 1979. The earliest less(1) I have was posted in mod.sources (comp.sources.unix) volume3, and the earliest entry in its changelog is January 1984.
- I initially thought more(1) was a pun on the AT&T pg(1) utility, but the 3BSD man page includes a reference to "the 'more' feature of the ITS systems at MIT," so the name may be related to the latter.

—Peter Jeremy, A FreeBSD Committer
(peter@rulingia.com)

MAY/JUNE

BY DRU LAVIGNE

2014 Events Calendar

These BSD-related conferences are scheduled for the second quarter of 2014. More information about these events, as well as local user group meetings, can be found at **bsdevents.org.**

LinuxTag • May 8–10, 2014 Berlin, Germany

http://linuxtag.org/2014/ • LinuxTag is Europe's leading conference and exhibition for professional users and developers of open source. There will be several BSD booths in the expo area and there are typically several BSD-specific presentations.

BSDCan • May 14–17, 2014 Ottawa, Canada

http://www.bsdcan.org/2014/ • The eleventh annual BSDCan will take place in Ottawa, Canada. This popular conference appeals to a wide range of people from extreme novices to advanced developers of BSD operating systems. The conference includes a Developer Summit, Vendor Summit, Doc Sprints, tutorials, and presentations. The BSDA certification exam will be available during the lunch break on May 16 and 17 and the first beta oF the BSDP lab exam will launch on May 18.

Texas LinuxFest • June 13–14, 2014 Austin, TX

http://texaslinuxfest.org/ • Texas LinuxFest is the first statewide, annual, community-run conference for Linux and open-source software users and enthusiasts from around the Lone Star State. There will be a BSD booth in the expo area and several BSD-themed presentations. The BSDA certification exam will be available on June 13.

SouthEast LinuxFest • June 20–22, 2014 Charlotte, NC

http://www.southeastlinuxfest.org/ • This is the sixth edition for this open source conference in the Southeast. There will be a BSD booth in the expo area and several BSD-related presentations. The BSDA certification exam will be available on June 22.